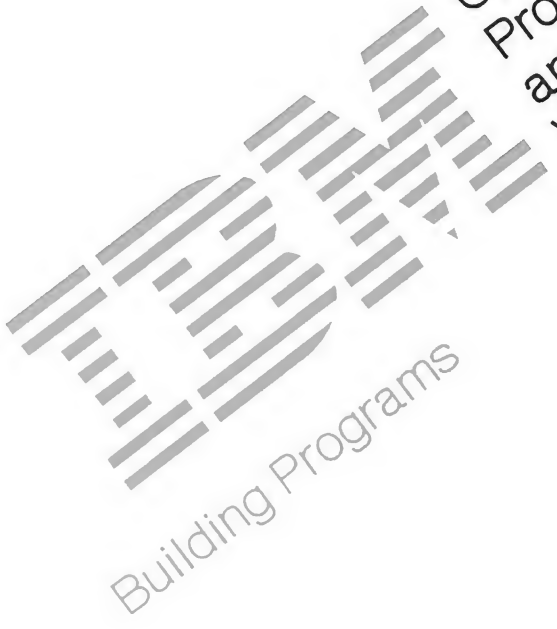

Operating System/2™ Programming Tools and Information Version 1.2

64F0274

Operating System/2™ Programming Tools and Information Version 1.2

Building Programs

First Edition (September 1989)

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Any reference to an IBM licensed program or other IBM product in this publication is not intended to state or imply that only IBM's program or other product may be used.

Requests for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

Note to US Government users — Documentation related to Restricted Rights — Use, duplication, or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

COPYRIGHT LICENSE: This publication contains printed sample application programs in source language, which illustrate OS/2 programming techniques. You may copy and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the OS/2 application programming interface.

Each copy of any portion of these sample programs or any derivative work, which is distributed to others, must include a copyright notice as follows: "© (your company name) (year) All Rights Reserved."

Special Notices

The following names, used in this publication, are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries:

IBM
C/2
COBOL/2
FORTRAN/2
Macro Assembler/2
Pascal Compiler/2
Operating System/2
OS/2
Presentation Manager

The following names, used in this publication, are trademarks or registered trademarks of other companies:

CodeView

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license enquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Armonk, NY 10504.

Preface

About This Book

The Operating System/2 Version 1.2 (OS/2) *Building Programs* book describes how to build Presentation Manager, Dialog Manager, text window, full-screen OS/2 and full-screen DOS applications.

This book also describes how to build a family application, a dynamic link library, an import library, and text-based message files. In addition, this book is a complete reference to the IBM Linker/2 (LINK) options, the module definition file statements, and the error codes generated by the utilities used to build applications.

Information in this book assumes you have written your source code following recommendations in the Operating System/2 Version 1.2 *Programming Guide*.

Prerequisite Knowledge

Building Programs is intended for professional application developers knowledgeable in at least one programming language in which OS/2 programs can be written, including:

- C/2 Version 1.1
- Macro Assembler/2 Version 1.0
- COBOL/2 Version 1.0
- FORTRAN/2 Version 1.02
- Pascal Compiler/2 Version 1.0
- BASIC Compiler/2 Version 1.0.

The information in the Programming Tools and Information library assumes you are new to programming with OS/2, the Presentation Manager, and Dialog Manager. You should understand the OS/2 services available to users. If you need help, consult the following books in the OS/2 product library:

- *Getting Started*
- *Using Advanced Features*
- *Product Information.*

The following information products can be purchased individually:

- *Command Reference*
- *Service Coordinator's Guide.*

Keyboards and Code Pages can be ordered separately at no cost.

Related Information

Programming Tools and Information

- *Installation*
- *Programming Overview*
- *Building Programs*
- *Programming Guide*
- *Control Program Programming Reference*
- *Presentation Manager Programming Reference Volume 1*
- *Presentation Manager Programming Reference Volume 2*
- *I/O Subsystems and Device Support Volume 1: Device Drivers*
- *I/O Subsystems and Device Support Volume 2: Presentation Driver Interface*
- *Dialog Manager Guide and Reference*
- *Dialog Tag Language Guide and Reference*
- *Dialog Manager and Dialog Tag Language Reference Summary*
- *Presentation Manager for C/2 Bindings Reference*
- *Presentation Manager for Macro Assembler/2 Bindings Reference*
- *Presentation Manager for FORTRAN/2 Bindings Reference*
- *Presentation Manager for COBOL/2 Bindings Reference*
- *Systems Application Architecture: Common User Access: Advanced Interface Design Guide.*

Systems Application Architecture Library

- *An Overview*
- *Writing Applications: A Design Guide*
- *Common User Access: Basic Interface Design Guide*
- *Common Programming Interface: C Reference*
- *Common Programming Interface: COBOL Reference.*
- *Common Programming Interface: FORTRAN Reference*
- *Common Programming Interface: Dialog Reference*
- *Common Programming Interface: Presentation Reference*

OS/2 and Related Product Libraries

OS/2 Product

IBM Operating System/2
Standard Edition
Version 1.2

Getting Started
Using Advanced Features
Product Information

6024926 3.5″ diskette
6024930 5.25″ diskette

OS/2 Programming Tools and Information

IBM Operating System/2
Version 1.2

Installation
Programming Overview
Programming Guide
Building Programs
Dialog Tag Language
Guide and Reference
Dialog Manager Guide
and Reference
Dialog Manager and
Dialog Tag Language
Reference Summary
Programming Reference
(3 books)
Bindings Reference for
Presentation Manager
(4 books for COBOL/2,
FORTRAN/2, C/2, and
Macro Assembler/2)
I/O Subsystems and
Device Support
(2 books)
Systems Application
Architecture
Common User Access:
Advanced Interface
Design Guide

6024929

Separate Order (no charge)

Keyboards and Code Pages

6280345

Available Separately

Command Reference

6024928

Service Coordinator's
Guide

15F2214

Programming Languages

IBM Basic Compiler/2
Version 1.0 6280179

IBM Macro Assembler/2
Version 1.0 6280181

IBM Pascal Compiler/2
Version 1.0 6280183

IBM FORTRAN/2
Version 1.02 6280185

IBM COBOL/2
Version 1.0 6280207

IBM C/2
Version 1.1 6280284

Systems Application Architecture

An Overview

GC26-4341

Writing Applications:
A Design Guide

SC26-4362

Common User Access:
Advanced Interface
Design Guide
SC26-4582

Common User Access:
Basic Interface Design
Guide
SC26-4583

Common Programming Interface

C Reference - Level 2

SC09-1308

COBOL Reference

SC26-4354

Dialog Reference

SC26-4356

FORTRAN Reference

SC26-4357

Procedures Language
Reference
SC26-4358

Presentation Reference

SC26-4359

Contents

Chapter 1. Introduction

Building Programs provides guidance in building executable files using the IBM Operating System/2™ Standard Edition, Version 1.2 (OS/2®[2]).[1] Information in this book assumes you have written one of these five types of applications: Presentation Manager, Dialog Manager, text window, full-screen OS/2 or full-screen DOS. This book also provides guidance in building a dynamic link library.

Do not read the entire book. Refer to the chapters you need to build the type of application you have written.

If you have not written your source code, refer to the *OS/2 Version 1.2 Programming Guide* for recommended programming techniques. The *Programming Guide* also describes which Application Programming Interface (API) calls can be used for each type of application. If you need information about OS/2 concepts and facilities, see the *OS/2 Version 1.2 Programming Overview*.

This book is organized to guide you in building an executable file or dynamic link library from the source code you have written. Each chapter begins with a figure illustrating the build process, which is described in more detail in the chapter.

Chapter 2, "Building a Presentation Manager Application" contains a detailed walk-through of the TEMPLATE sample program in the tools package.

The walk-through includes the following:

- Language specific compiler options
- Module definition file statements
- OS/2 Linker (LINK) options
- Resources and resource compiling using the Resource Compiler (RC) utility
- Resources in dynamic link libraries.

The chapter also tells you how to modify the source code files so the TEMPLATE sample program can use resources in a dynamic link library.

Chapter 3, "Building a Dialog Manager Application" contains the elements and commands to build Dialog Tag Language files. This chapter also includes:

- Language specific compiling options for the Dialog Manager source code
- LINK options
- Module definition file statements.

This chapter refers to the DMDEMO sample program available in C/2 and COBOL/2 in the tools package.

Chapter 4, "Building Text Window and Full-Screen Applications" describes how to build the executable file for the text window, full-screen OS/2, and full-screen DOS applications. The chapter also describes:

- Binding the application, using the BIND utility to create a family application that can execute in the OS/2 or DOS environments.
- Creating a text message file using the MKMSGF utility. The message file is separate from the application's executable file, making the messages easy to translate into foreign languages. The optional MSGBIND utility binds the file to the application's executable file.

This chapter refers to the VIOSAMPC.C sample program in the \TOOLKT12\C\SAMPLES\BSE\VIOCBAT subdirectory, and WTC.C sample program in the \TOOLKT12\C\SAMPLES\BSE\WTCBAT subdirectory. The VIOSAMPC.C sample program is written using the subset of family API calls, so it can be compiled, linked, and bound (using the BIND utility) to run in both the OS/2 and DOS environments. The WTC.C sample program contains the calls needed to display a text message in any non-Presentation Manager application.

---

[1] Trademark of IBM Corporation

[2] Registered Trademark of IBM Corporation

Chapter 5, "Building a Dynamic Link Library" contains a detailed walk-through of the TYPETEXT sample program. The chapter guides you in compiling and building a dynamic link library, including:

- Compiler options
- Module definition file statements
- LINK options
- Import libraries created using the IMPLIB utility.

Chapter 6, "The LINK Utility" is a complete guide to using LINK. You can respond to LINK prompts, type a command at the OS/2 command prompt, or create a response file. Although the documentation for LINK is in this programming library, the LINK utility is packaged with the *Operating System/2* Version 1.2 product.

Chapter 7, "LINK Options," Chapter 8, "Module Definition File Statements" and **Appendix A, "Error Messages"** contain comprehensive reference information for the following subjects:

- LINK options
- Module definition file statements
- Error messages produced by the LINK, RC, BIND, IMPLIB, MKMSGF, and MSGBIND utilities.

BIND, IMPLIB, MSGBIND, and RC utilities are supplied on the tools diskettes. The WINABLE utility is also supplied on the tools diskettes, and contains information necessary to convert OS/2 full-screen applications to run in a Presentation Manager window. Refer to the *Programming Guide* for detailed information on the WINABLE utility.

Chapter 2. Building a Presentation Manager Application

This chapter describes how to build a Presentation Manager application. It shows how to compile the source code and resources, and how to link them using the OS/2 Linker (LINK). The TEMPLATE sample program is used to demonstrate the build process. TEMPLATE is a C/2 program, but the only stage in the build process where C/2, COBOL/2, and FORTRAN/2 programs differ is in compilation. For this stage, examples are given for each language. The source files for the TEMPLATE sample are located in the \TOOLKT12\C\SAMPLES\PM\TEMPLATE subdirectory.

The Build Process

To build a Presentation Manager application, you must first create source code following the programming techniques described in the *Programming Guide*. You must then compile or assemble the source code, and finally link the object modules together to generate an executable file.

In addition, your application may use resources such as menus, accelerator tables, string tables, dialog boxes, icons, pointers, bit maps, fonts, help panels, and help tables. These are also described in the *Programming Guide*. You use a text editor to create a file called a resource script (.RC) file that is separate from your source code. It defines menus, accelerator tables, string tables, and help tables using resource file statements whose syntax is described in the *Presentation Manager Programming Reference Volume 2*. The .RC file is an input file for the RC utility, a resource compiler that converts all the resources into an executable form. In addition to containing statements that explicitly define resources, the resource file can contain statements that supply the resource compiler with the names of files containing other resources.

You can use the resource editors to define dialog boxes, icons, pointers, bit maps, and fonts. The resource editors are the Dialog Box Editor, the Icon Editor, and the Font Editor. They are supplied with the programming tools. They generate output files with the following file name extensions:

Dialog Box	.DLG
Icon	.ICO (icon)
	.PTR (pointer)
	.BMP (bit map)
Font	.FNT

You can supply the file names of icons, pointers, bit maps, and dialog boxes in the same resource file in which you define menus, accelerator tables, string tables, and help tables.

Dynamic linking is available in C/2 and Macro Assembler/2. All resources except fonts can be bound to the application's executable file or compiled into a dynamic link library. This process is described in "The Build Process for Application Resources" on page 2-5. Fonts must be put in separate dynamic link libraries. This technique is described in "Putting Resources into a Dynamic Link Library" on page 2-9.

Note: Binding resources using the RC utility is *not* related to binding an executable file using the BIND utility. BIND is discussed in Chapter 4, "Building Text Window and Full-Screen Applications."

Generating the Executable File

To build the executable file for an application that has no resources:

1. Compile or assemble the source code using the C/2 compiler, COBOL/2 compiler, FORTRAN/2 compiler, or Macro Assembler/2 assembler. This creates object (.OBJ) files.
2. Write the module definition (.DEF) file.
3. Link the object files with the module definition file and any library (.LIB) files. This creates the executable (.EXE) file.

Figure 2-1 illustrates the build process for an executable file without resources.

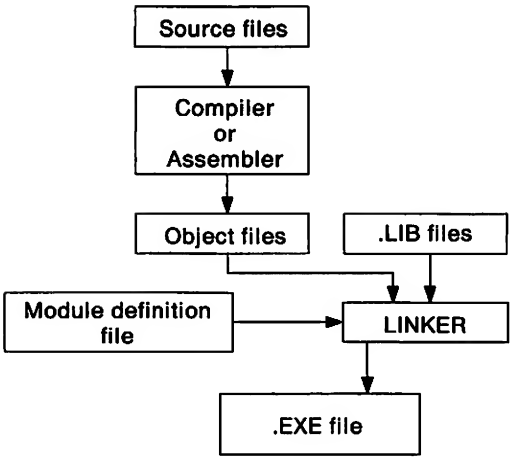

Figure 2-1. Building the Application's Executable File

Compiling C/2 Source Code

The TEMPLATE sample application is built using the TEMPLATE.CMD file. The compiler commands are in the *make* file, TEMPLATE.MAK, which is called from the TEMPLATE.CMD file. For further information on .CMD files, see *Using Advanced Features*.

The case-sensitive compiler command for each of the source code (.C) files is:

```
cl /c /Alfu /W3 /Gs /Gc filename.c
```

where:

/c specifies compile only (link is a separate step)

/Alfu produces far code and data pointers, as in the C large-memory model, and loads the DS segment register on entry to your routine.

/W3 requests level 3 warning messages

/Gs turns off stack checking

/Gc requests the Pascal calling convention for the subroutines.

The compiler options you specify must be compatible with the Presentation Manager. Note that because the DS register must be loaded on entry to any window or dialog procedure, your memory model (/A) option must include /Au.

Your application might not require a far pointer for every reference to data. So you can specify near data pointers using the /An option in place of /Af. However, the C compiler issues warning C4058 for any reference to a frame variable with a near pointer. An example is passing the address of a local array to a routine in a C run-time library. The warning should be heeded unless you can guarantee that stack segment and data segment are equal at the time the reference to the data is made.

Other C/2 Compiler Considerations

The compiler options in this chapter are specific to the sample program that is used to illustrate the build process. To ensure that you are able to compile, link, and run the sample programs, a command (.CMD) file is supplied for each program. The command file sets the environment for the compile and link process, and calls a make (.MAK) file that contains the compiler and LINK commands.

The commands for each sample application are not the same, so this file is especially helpful. For instance, the example in this chapter uses the /Alfu option, which does not require EXPORT statements. If you use the /Gw option, your module definition (.DEF) file must contain the EXPORT statement for each procedure registered. If you do not use the /Gw option and the EXPORT statements, the application will fail.

The advantage of the /Alfu option is that the module definition (.DEF) file is independent of the code. If you use the /Aw and /Gw options, the module definition file is required to contain all of the window procedures in the EXPORTS statements. If you should add or delete the Window Procedures, then you must add or delete the corresponding EXPORTS statements.

If you elect to use the /Alfu option, the module definition file can be defined once. However, if the structure of your application changes and requires a bigger stack or heap, you would have to change the module definition (.DEF) file whether you were using the /Gw option or the /Alfu option.

For additional information on Presentation Manager compiler options, see *IBM C/2, Version 1.1*.

Compiling COBOL/2 Source Code

The DLG1COB COBOL sample application is compiled with the following case-sensitive command:

```
pcobol dlg1cob,, /confirm /ans85
/litlink /model"LARGE";
```

where:

/confirm
: lists the entered directives on the screen during compilation

/ans85
: uses reserved words specific to ANSCOBOL 85, and enables ANSCOBOL 85 file handling and status codes

/litlink
: creates external references for all CALL *literal* statements, where *literal* is a public symbol corresponding to the name of a COBOL or non-COBOL procedure

/model"LARGE"

- produces external references for CALL *literal* statements, when you have specified /litlink
- produces FAR calls for all CALL *literal* statements
- produces 32-bit pointers as BY REFERENCE parameters for all CALL *literal* statements.

DLG1COB is built using the DLG1COB.CMD file. The compiler commands are in the *make* file, DLG1COB.MAK, which is called from the DLG1COB.CMD file. For further information on .CMD files, see *Using Advanced Features*.

Compiling FORTRAN/2 Source Code

The DLG1FOR FORTRAN sample application is compiled with the following case-sensitive command:

```
fortran dlg1for /N
```

where /N creates code that does not depend on the presence of the math coprocessor.

DLG1FOR is built using the DLG1FOR.CMD file. The compiler commands are in the *make* file, DLG1FOR.MAK, which is called from the DLG1FOR.CMD file. For further information on .CMD files, see *Using Advanced Features*.

Writing the Module Definition File

The module definition (.DEF) file is an ASCII text file that you can create with any suitable text editor. An example of a module definition file is provided in the \TOOLKT12\C\SAMPLES\PM\TEMPLATE subdirectory and is illustrated in Figure 2-2.

```
NAME    Template  WINDOWAPI

DESCRIPTION 'OS/2 Presentation Manager Application Template'

STUB    'OS2STUB.EXE'

DATA    MULTIPLE

STACKSIZE   4096        ; recommend 4k as minimum stacksize
HEAPSIZE    4096        ; recommend 4k as minimum heapsize

PROTMODE
```

Figure 2-2. TEMPLATE.DEF Sample Module Definition File

The module definition file statements tell LINK about the executable file it is creating. The statements used in the TEMPLATE.DEF sample program are:

NAME
: tells LINK that the executable file being created is a program module called TEMPLATE. WINDOWAPI sets a flag in the executable file's header meaning that this program must execute in a Presentation Manager window. This statement is mandatory.

DESCRIPTION
: inserts a text description into the executable file.

STUB
: calls a stub file that generates an error message if the user attempts to execute this Presentation Manager application in the DOS environment. OS2STUB.EXE is a DOS executable file that is added to the beginning of the program module. This file is provided in the \TOOLKT12\BIN subdirectory.

DATA
: defines the attributes of all data segments in TEMPLATE.

STACKSIZE
: defines the stack size; a minimum of 4096 bytes is recommended for Presentation Manager applications.

HEAPSIZE
: defines the heap size; a minimum of 4096 bytes is recommended for Presentation Manager applications.

PROTMODE
: specifies that the executable file can run only in the OS/2 environment.

For a complete list of the module definition file statements and their options, see Chapter 8, "Module Definition File Statements."

Linking the Object Files

To generate your application's executable file, use the LINK utility that is supplied with OS/2.

Note: The OS/2 LINK utility is **not** the same as the utility supplied with any language-specific software.

You need the following files:

Object Files
: The object (.OBJ) files are created by your language compiler or assembler. You must compile or assemble all application source code files.

Library Files
: LINK resolves intersegment calls using library (.LIB) files. If you have called external subroutines or subroutines in dynamic link libraries, you must link the library files together with the object files. You can specify a library search path using the SET LIB=*path* command before linking.

Module Definition File
: This file tells LINK about the executable file it is creating.

The file called TEMPLATE.L, located in the \TOOLKT12\C\SAMPLES\PM\TEMPLATE subdirectory, contains responses to the LINK prompts. Figure 2-3 shows the contents of this file.

```
tempinit.obj+
tempnres.obj+
tempres.obj     /A:16
template.exe
template.map    /NOD
llibce.lib+
os2.lib
template.def
```

Figure 2-3. TEMPLATE.L Sample Response File

The TEMPLATE sample program consists of three object (.OBJ) files. The responses tell LINK to generate the TEMPLATE.EXE executable file from the object files called TEMPINIT.OBJ, TEMPNRES.OBJ, and TEMPRES.OBJ and to create a map file called TEMPLATE.MAP. LINK sequentially searches the library files LLIBCE.LIB and OS2.LIB to resolve intersegment references and uses the module definition file called TEMPLATE.DEF.

The /A:16 LINK option specifies alignment 16, which is recommended for Presentation Manager applications.

The /NOD option tells LINK to ignore any default libraries, for example DOSCALLS.LIB.

To generate the application's executable file, type this command at the OS/2 command prompt:

```
link @template.l
```

The executable file contains references to resources defined in a separate resource file. For TEMPLATE, this is TEMPLATE.RC shown in Figure 2-4 on page 2-6.

The Build Process for Application Resources

Most Presentation Manager applications use resources. These can be menus, accelerator tables, string tables, dialog boxes, icons, pointers, bit maps, fonts, and help panels.

Resource Script File

All resources are defined in a resource script (.RC) file. Fonts have a resource script file to themselves, as described in "Font Resources" on page 2-11. Resources are either defined explicitly in statements in the resource script file, or they are defined in other files (such as output files from the resource editors) that are referenced in the resource script file. For example, the statement:

```
rcinclude template.dlg
```

includes the dialog (.DLG) file created using the Dialog Box Editor.

The resource script (.RC) file is compiled using the Resource Compiler (RC) utility to generate a .RES file. The compiled resource (.RES) file must have the same name as the application's executable file.

The resource script file for TEMPLATE is shown in Figure 2-4.

```
#define INCL_WINHELP
#include <os2.h>
#include "template.h"
#include "tempdlg.h"

HELPTABLE ID_TEMPLATE
BEGIN
    HELPITEM ID_TEMPLATE, ID_TEMPLATE, IDH_TEMPLATE_EXT
    HELPITEM ID_EXAMPLE,  ID_EXAMPLE,  IDH_EXAMPLE_EXT
    HELPITEM IDSABOUT1,   ID_MESSAGE,  IDH_ABOUT_EXT
END
HELPSUBTABLE ID_TEMPLATE
BEGIN
    HELPSUBITEM IDMEXAMPLE,      IDH_EXAMPLE
    HELPSUBITEM IDMDIALOG,       IDH_DIALOG
    HELPSUBITEM IDMEXIT,         IDH_EXIT
    HELPSUBITEM IDMHELP,         IDH_HELP
    HELPSUBITEM IDMHOWTO,        IDH_HOWTO
    HELPSUBITEM SC_HELPEXTENDED, IDH_HELPEXTENDED
    HELPSUBITEM SC_HELPINDEX,    IDH_HELPINDEX
    HELPSUBITEM SC_HELPKEYS,     IDH_HELPKEYS
    HELPSUBITEM IDMABOUT,        IDH_ABOUT
END
HELPSUBTABLE ID_EXAMPLE
BEGIN
    HELPSUBITEM IDDLISTBOX, IDH_LISTBOX
    HELPSUBITEM DID_OK,     IDH_OK
    HELPSUBITEM DID_CANCEL, IDH_CANCEL
    HELPSUBITEM IDDHELP,    IDH_HELPBUTTON
END
HELPSUBTABLE ID_MESSAGE
BEGIN
END

ICON ID_TEMPLATE   template.ico

MENU ID_TEMPLATE   PRELOAD
BEGIN
    SUBMENU  "~Example",                          IDMEXAMPLE
    BEGIN
        MENUITEM "~Dialog Example...\tCtrl+D",    IDMDIALOG
        MENUITEM SEPARATOR
        MENUITEM "E~xit \tF3",                    IDMEXIT
    END
    SUBMENU  "~Help",                             IDMHELP
    BEGIN
        MENUITEM     "~Help for help...",         IDMHOWTO
        MENUITEM     "~Extended help...",         SC_HELPEXTENDED, MIS_SYSCOMMAND
        MENUITEM     "~Keys help...",             SC_HELPKEYS, MIS_SYSCOMMAND
        MENUITEM     "Help ~Index...",            SC_HELPINDEX, MIS_SYSCOMMAND
        MENUITEM     SEPARATOR
        MENUITEM     "~About...",                 IDMABOUT
    END
END
```

Figure 2-4 (Part 1 of 2). TEMPLATE.RC Sample Resource Script File

```
ACCELTABLE      ID_TEMPLATE
BEGIN
    VK_F3,      IDMEXIT,    VIRTUALKEY
    "^D",       IDMDIALOG
END

STRINGTABLE
BEGIN
    IDSNAME,        "Template"
    IDSSTR1,        "Windermere"
    IDSSTR2,        "Coniston Water"
    IDSSTR3,        "Ullswater"
    IDSSTR4,        "Bassenthwaite Lake"
    IDSSTR5,        "Buttermere"
    IDSSTR6,        "Derwent Water"
    IDSSTR7,        "Thirlmere"
    IDSTEXTSTR1,    "Example Text - "
    IDSTEXTSTR2,    "Center Screen"

    IDSABOUT1,      "           OS/2 Presentation Manager"
    IDSABOUT2,      "                    Release 1.2"
    IDSABOUT3,      "             Template Sample Program"
    IDSABOUT4,      "      (C) Copyright IBM Corp. 1988, 1989"
    IDSABOUT5,      "      (C) Copyright Microsoft Corp. 1988, 1989"
END

rcinclude template.dlg
```

Figure 2-4 (Part 2 of 2). TEMPLATE.RC Sample Resource Script File

The #define statement in the first line of the file includes the help manager definitions file. Next there are three include files:

os2.h
: This OS/2 header file, located in the \TOOLKT12\C\INCLUDE subdirectory, resolves defined system variables and constants.

template.h
: This file is created as part of the application to resolve the variables and constants within the TEMPLATE code modules.

tempdlg.h
: This file is generated by the Dialog Box Editor to resolve the variables and constants used within the dialog template.

The resource script file defines the resources next. Help panels, icons, menus, accelerator tables, text strings, and dialogs are used in this application. They are identified by these statements in the resource script file:

HELPTABLE ID_TEMPLATE
: links the helpsubtable and extended help panels with the item that they are describing.

HELPSUBTABLE
: defines the field-level help for each window. It links an item with the ID of the help panel that describes it. The third subtable, ID_MESSAGE, is for help on message boxes.

ICON ID_TEMPLATE
: references the template.ico file, created using the Icon Editor.

MENU ID_TEMPLATE
: contains menu items within the BEGIN and END statements.

ACCELTABLE ID_TEMPLATE
: contains accelerator keys within the BEGIN and END statements.

STRINGTABLE
: contains text strings within the BEGIN and END statements.

rcinclude template.dlg
: references the template.dlg file, created using the Dialog Box Editor.

Binding Resources to the Application

An application can access resources in two ways:

- By binding the resources to the application's executable file.
- By putting the resources into a dynamic link library.

The TEMPLATE sample program uses the first method. The method you use depends on how you wrote the application. There are different build procedures and application considerations for each method. The second method is described in "Rebuilding TEMPLATE.C for Resources in a Dynamic Link Library " on page 2-11. It shows the changes needed in the TEMPLATE source code files to build the TEMPLATE resources into a dynamic link library.

To bind the resources to the application's executable file, you can do either of the following:

- Compile and bind the resources in the resource script (.RC) file to the application's executable file by running the RC utility in one step.
- Compile and bind the resources in two steps. First, run the RC utility to compile the resource script (.RC) file. This process generates a file with a .RES file name extension. Then, bind the .RES file to the application's executable file by running the RC utility again.

The second method is useful when developing applications in which the source code files are likely to change but the resources will remain unchanged. In such cases, the resource script file needs to be compiled only once and then bound to the application's executable file each time the application is rebuilt.

Compiling and Binding Resources in One Step

Figure 2-5 illustrates the process:

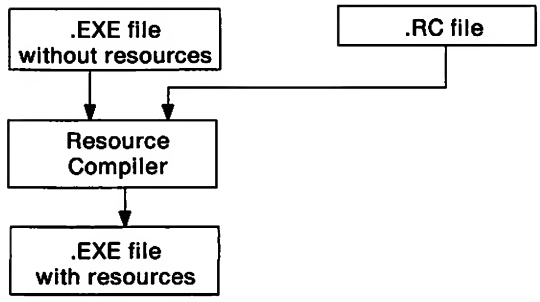

Figure 2-5. Compiling and Binding Resources

To build the TEMPLATE sample program resources in one step, type the following command at the OS/2 command prompt:

```
rc template.rc
```

This command compiles the TEMPLATE.RC script file and binds the output to TEMPLATE.EXE. Note that this file has the resources appended to it. If the resource script file is changed, it can be recompiled and bound again to the application's executable file. The original resources are replaced by the new ones.

Compiling and Binding Resources in Two Steps

Figure 2-6 illustrates the process:

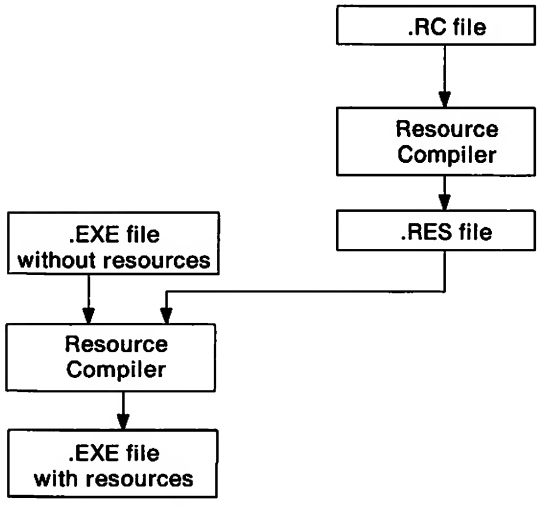

Figure 2-6. Compiling and Binding Resources in Separate Operations

To build the TEMPLATE sample program in two steps, first type the following command at the OS/2 command prompt to compile the resource file:

```
rc -r template.rc
```

The -r option tells the resource compiler to save the compiled resource file as a .RES file.

Then type the command:

```
rc template.res
```

This binds the compiled resource (.RES) file to the application's executable file, TEMPLATE.EXE.

Starting the Resource Compiler

To start the RC utility, type the following command at the OS/2 command prompt:

```
rc [ -r ] infile [ outfile ]
```

where:

-r
: tells the resource compiler to compile the resource script (.RC) file and not bind the resulting compiled resource (.RES) file to the application's executable file.

infile
: is the name of the:

- Resource script (.RC) file if you are compiling only, or compiling and binding in one step.
- Compiled resource (.RES) file if you are binding. The default file name extension is .RC, so you must specify .RES as the extension.

outfile
: is the new name of the:

- Compiled resource (.RES) file or dynamic link library (.DLL) file if you are compiling.
- Application's executable (.EXE) file if you are binding, or compiling and binding in one step.

If you do not specify this option, the default file name is that used for **infile**.

The error messages produced by the RC utility are listed in Appendix A, "Error Messages."

Putting Resources into a Dynamic Link Library

Instead of binding a resource file to your application, you can put it into a dynamic link library. You then link the file at run-time and load the resources into your application by using the DosLoadModule or GpiLoadFonts calls.

Note that you cannot switch from binding resources to putting resources into a dynamic link library without changing your application source code. Figure 2-7 illustrates how to put resources into a dynamic link library.

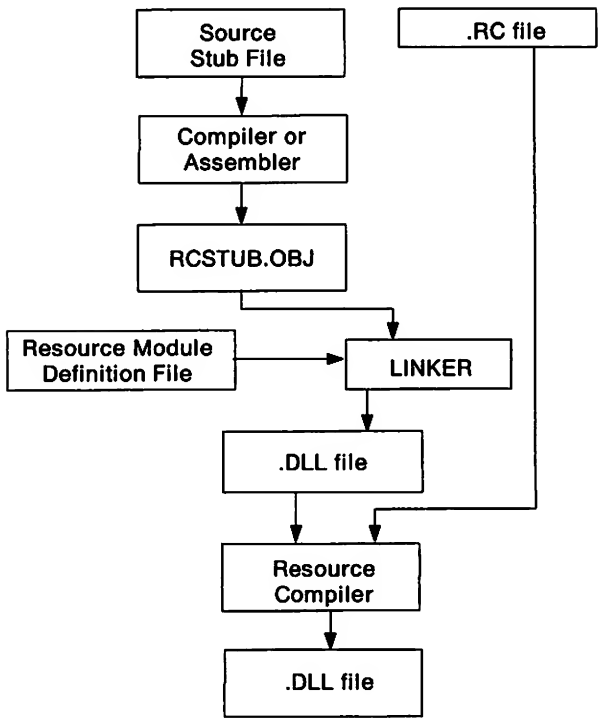

Figure 2-7. Building a Dynamic Link Library for Resources

Follow these steps:

1. Create the resource script (.RC) file that defines the resources, as described in "The Build Process for Application Resources" on page 2-5.
2. Create a source stub file in C/2 or Macro Assembler/2, as shown in "Stub File for a Dynamic Link Library."
3. Compile or assemble the source stub file to create the RCSTUB.OBJ file.
4. Create a module definition (.DEF) file, as shown in Figure 2-8.
5. Link the RCSTUB.OBJ file with the module definition file (.DEF) to create the dynamic link library (.DLL) file for the resources.
6. Compile the resource script (.RC) file and incorporate the resources into the dynamic link library (.DLL) using the resource compiler.

7. Copy the resulting dynamic link library (.DLL) file into the appropriate directory for use by the application. The file must be in one of the directories specified in the library search path. This is defined by the LIBPATH statement in the CONFIG.SYS file.

Stub File for a Dynamic Link Library

The resource stub file contains no data or executable code. It is simply a framework on which to build the dynamic link library. The stub file can be written in C/2 or Macro Assembler/2.

The source code for RCSTUB.C is:

```
int _acrtused=1;

void far pascal RCSTUB()
{
}
```

To compile this source code, type the following command at the OS/2 command prompt:

```
cl /c /Gs rcstub.c
```

The source code for RCSTUB.ASM is:

```
TITLE rcstub.asm

CODE segment

CODE ends

end
```

To assemble this source code, type the following command at the OS/2 command prompt:

```
masm rcstub.asm
```

Module Definition File for a Dynamic Link Library

The module definition (.DEF) file is an ASCII text file that can be used to tell LINK to create a dynamic link library. Figure 2-8 illustrates a sample module definition file for creating a dynamic link library.

```
LIBRARY RESOURCE

DESCRIPTION 'DLL file containing resources'

STUB 'OS2STUB.EXE'

DATA NONE
```

Figure 2-8. RESOURCE.DEF Sample Module Definition File

The name RESOURCE in the LIBRARY statement is the name of the dynamic link library that contains the resource file specified with the DosLoadModule or GpiLoadFonts calls. If you change the name of the dynamic link library without changing the module definition file, the load calls generate an error.

Creating a Dynamic Link Library for Resources

To set the environment for the compile and link process, call the PMPATH.CMD command file. This is demonstrated in the TEMPLATE.CMD file.

The dynamic link library resource stub file is created by linking the resource stub object file with the resource module definition file. To start the LINK utility, type the following command on a single line at the OS/2 command prompt:

```
link rcstub.obj, resource.dll,
                        nul, /NOD, resource.def
```

where:

nul — specifies that the map file is not required

/NOD — tells LINK to ignore any default library files, for example DOSCALLS.LIB.

Alternatively, you can put the commands in a response file (RESOURCE.L, for example) which you invoke by typing the following command:

```
link @resource.l
```

LINK takes the RCSTUB.OBJ file, together with the RESOURCE.DEF file, and creates a file called RESOURCE.DLL.

Finally, the resource script file must be compiled and incorporated into the dynamic link library

resource file. To start the RC utility, type the following command at the OS/2 command prompt:

```
rc resource.rc resource.dll
```

The RC utility compiles the RESOURCE.RC file and puts the results in a file called RESOURCE.DLL.

The dynamic link library (.DLL) file must be copied to the correct directory so the application issuing the DosLoadModule call can access it. See the LIBPATH command in the *Command Reference* for more information.

Font Resources

Resource files can contain either font resources or a mixture of the other types of resources. Do not mix fonts with other types of resources because fonts must be loaded using the GpiLoadFonts call. All other resources are loaded using the DosLoadModule call.

A dynamic link library containing font resources must have a file name extension of .FON.

Therefore, when you have created the .DLL as described for other resources, you must rename it when you copy it to the correct directory.

Rebuilding TEMPLATE.C for Resources in a Dynamic Link Library

For the TEMPLATE sample program to use resources in a dynamic link library, the source code files TEMPINIT.C, TEMPNRES.C, and TEMPRES.C need modifying.

The changes to the TEMPINIT.C source code file are shown in Figure 2-9. The #define constant INCL_DOSMODULEMGR and the DosLoadModule call have been added. RCHandle is global to all three files. The modified TEMPNRES.C and TEMPRES.C source code files are shown in Figure 2-10, and Figure 2-11. All changes and additions to the source code are shown in **bold** print.

```
#define INCL_WINFRAMEMGR
#define INCL_DOSMODULEMGR
#include <os2.h>
#include "template.h"

extern HAB     hab;
extern HWND    hwndFrame;
extern HWND    hwndClient;
extern HMODULE RCHandle;

extern char    szWindowTitle[];
     .
     .
BOOL FAR TemplateInitApp( VOID )
{
    char szClassName[10];
    FRAMECDATA fcdata;

    DosLoadModule( NULL, 0, "RESOURCE", RCHandle );

    WinLoadString( hab, &RCHandle, IDSNAME, sizeof(szClassName), (PSZ)szClassName );

    if (!WinRegisterClass( hab,
                        (PSZ)szClassName,
                           .
                           .
                           .
    fcdata.cb            = sizeof(FRAMECDATA);
    fcdata.flCreateFlags = FCF_STANDARD;
    fcdata.hmodResources = RCHandle;
    fcdata.idResources   = ID_TEMPLATE;

    hwndFrame = WinCreateWindow(HWND_DESKTOP,
                                WC_FRAME,
                                (PSZ)NULL,
                                0L,
                                0,0,0,0,
                                NULL,
                                HWND_TOP,
                                ID_TEMPLATE,
                                &fcdata,
                                NULL);
                                    .
                                    .
    hinit.hmodHelpTableModule = RCHandle;
                                    .
                                    .
    WinLoadString(hab, RCHandle, IDSTEXTSTR1, CCHMAXSTRING, (PSZ)szText);
    WinLoadString(hab, RCHandle, IDSTEXTSTR2, CCHMAXSTRING, (PSZ)szExampleText);

    return(TRUE);
}
```

Figure 2-9. Modified TEMPINIT.C source code file

```
#define INCL_GPI
#define INCL_WIN

#include <os2.h>
#include <string.h>
#include "template.h"
#include "tempdlg.h"

extern HAB     hab;
extern HWND    hwndFrame;
extern HWND    hwndClient;
extern HMODULE RCHandle;
extern char    szText[];
                .
                .
                .
          sOff += WinLoadString( hab,
                                 RCHandle,
                                 sCounter,
                                 sizeof(szTemp)-sOff,
                                 &szTemp[sOff]
                               );
                .
                .
                .
        WinLoadString(hab, RCHandle, IDSSTR1+i, CCHMAXSTRING , (PSZ)szWaters);
        WinSendDlgItemMsg( hwndDlg,
                .
                .
                .
```

Figure 2-10. Modified TEMPNRES.C source code file

```
#define INCL_WIN

#include <os2.h>
#include "template.h"
#include "tempdlg.h"

HAB     hab;
HWND    hwndFrame;
HWND    hwndClient;
HMODULE RCHandle;
char    sztext[CCHMAXSTRING];
                .
                .
                .
               case IDMDIALOG:
                 WinDlgBox(HWND_DESKTOP,
                           hwndFrame,
                           (PFNWP)TemplateDlg,
                           RCHandle,
                           ID_EXAMPLE,
                           (PCH)NULL);
                 break;
                   .
                   .
                   .
  return(MRFROMLONG(NULL));
}
```

Figure 2-11. Modified TEMPRES.C source code file

If you modify the source code and recompile the three files, the TEMPLATE program will execute providing you put its resources into a dynamic link library. The dynamic link library must be in a directory that is defined in the LIBPATH statement of the CONFIG.SYS file. Also, you must remove from the TEMPLATE.MAK file all references to the TEMPLATE.RES file. The modified make file is shown in Figure 2-12.

```
template.hlp: template.itl
        itlc template.itl

tempinit.obj: tempinit.c template.h
        cl /c /Alfu /W3 /Gs /Gc tempinit.c

tempnres.obj: tempnres.c template.h
        cl /c /Alfu /W3 /Gs /Gc tempnres.c

tempres.obj: tempres.c template.h
        cl /c /Alfu /W3 /Gs /Gc tempres.c

template.exe: template.l tempinit.obj tempnres.obj tempres.obj  \
              template.def
        link @template.l
```

Figure 2-12. Modified TEMPLATE.MAK make file

Loading Resources

To load resources other than fonts, use the DosLoadModule call. This call returns a handle which you use as an input parameter to the following API calls:

GpiLoadBitMap
WinLoadAccelTable
WinLoadDlg
WinDlgBox
WinLoadMenu
WinLoadPointer
WinLoadString
WinCreateStdWindow
WinCreateFrameControl

Alternatively, you can use the handle as part of the FRAMECDATA structure that is used with the WinCreateWindow call.

If you do not put the resources in a dynamic link library, specify a NULL value for the handle.

Packaging Presentation Manager Applications

When your application is ready for delivery, you must remember to ship all its components. These include the executable file, any separate resources, and all dynamic link libraries.

Chapter 3. Building a Dialog Manager Application

This chapter describes how to build a Dialog Manager application which consists of two build processes. The first build creates the Dialog Tag Language (.DTL) files created by the Dialog Tag Language (DTL) compiler. If your application contains Help panels, .HLP files also are created. The final build is the creation of the executable (.EXE) file. These files are created by any of the OS/2 supported compilers. Figure 3-1 illustrates the two build processes:

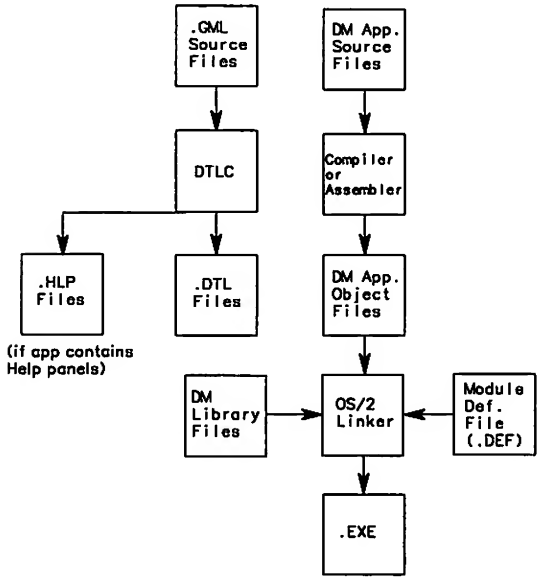

Figure 3-1. Dialog Tag Language Files and the Application's Executable File

Building the Dialog Tag Language Files

This section contains a brief description of the Dialog Tag Language Compiler (DTLC). For a complete description see *Dialog Tag Language Guide and Reference*.

After you have finished coding dialog elements in your source files, the DTL compiler creates the elements and stores them in a DTL or HLP library file. HLP library files store the help panels that the Information Presentation Facility (IPF) displays under the direction of the Dialog Manager. DTL library files are used to store other dialog elements managed by the Dialog Manager.

DTL provides you with commands to perform the steps necessary for preparing dialog elements used in an application. These commands are:

DTLC To verify tag syntax and create formatted DTL and HLP library files.

DTLD Tp delete and remove dialog elements from a DTL library. DTLD is not used for HLP libraries.

DTL also provides you with a display utility that allows you to view the formatted application panels and messages in your files, without requiring an underlying Dialog Manager program.

In this chapter, we show you how to use the DTL commands to prepare your files for use in Dialog Manager applications. We also show you how to use the DISPLAY utility to list the formatted elements in a DTL library and see how panels and messages appear when displayed in an application.

Executing a Dialog Manager Sample Application

Compile the source code to generate object (.OBJ) files.

When using the C/2 compiler, the following command will compile, link and execute the Dialog Manager sample application.

```
DMDEMO.CMD
```

When using the COBOL/2 compiler, the following command will compile, link and execute the Dialog Manager sample application.

```
DMDEMOCO.CMD
```

Compiling and Verifying Source Files

The DTLC command compiles the dialog elements in your source files. You can also use the DTLC command to verify the syntax in your source files. You perform the syntax check as a separate operation, or you may perform this syntax check when you actually compile the source file.

Verifying Markup Syntax

If you want to verify the markup syntax in a source file and display the errors before you compile it, issue the DTLC command with the /SYNTAX option. When you specify the /SYNTAX option, the compiler does not create a file for the dialog elements in the source file.

For example, to check the syntax in a source file named MYTAGS.GML, issue the following command:

```
dtlc mytags /syntax
```

If you use the minimum abbreviation of /SYNTAX, issue the same command this way:

```
dtlc mytags /s
```

Compiling Source Files

To compile a source file, use the DTLC command without the /SYNTAX option. For example,

```
dtlc mytags
```

In this case, the compiler not only checks the syntax of the source file and issues messages for errors and warnings, but it also formats the dialog elements within the source file and writes them to libraries named MYTAGS.DTL and MYTAGS.HLP.

If the compiler encounters an error, it stops writing the elements to the DTL library and does not create (or replace) the HLP library, although it continues to verify the syntax. If the compiler encounters a severe error, it halts all processing and exits.

Specifying a Library

Following the completion of the compile operation, the compiler produces DTL library files as output. The compiler adds the extension .HLP to the file name to identify the file as a library containing help panels. The compiler adds the extension .DTL to the filename to identify the file as a library containing other dialog elements. For example, the library files created for the dialog elements in the source file MYTAGS.GML are named MYTAGS.DTL and MYTAGS.HLP.

The compiler creates a separate library file for the command table, and names the file by appending the string CMDS to the application name you specify for the command table. The compiler also adds the .DTL extension to the file name.

For example, if the application command table defined within the MYTAGS.GML file has the application name XYZ, the compiler creates the libraries MYTAGS.DTL, MYTAGS.HLP, and XYZCMDS.DTL.

If you want use a name other than the source file name for the output libraries, you can specify this choice with the LIBSPEC parameter of the DTLC command. For dialog elements stored in DTL libraries, you can place the elements defined by multiple source files within a single library by specifying the same LIBSPEC value when compiling each of the source files.

Note: Help Panel (HLP) library files are created or replaced in their entirety each time the DTL compiler is called. You should be careful when specifying the LIBSPEC parameter that an existing HLP library is not being overwritten.

For example, suppose the source file MYTAGS.GML contains the definitions for some of the application panels, messages, and key mapping lists for your application, and the source file MORETAGS.GML contains other definitions. The elements in both of these source files may be compiled and stored within the same library named ALLTAGS.DTL by issuing the following DTLC commands:

```
dtlc mytags,alltags
dtlc moretags ,alltags
```

You must separate the library specification from the source file name with a comma. In this example, we did this in two different ways. In the first DTLC call, we used only a comma. In the second DTLC call, we added a blank before the comma. We also could have added more blanks—it doesn't matter because the DTL compiler is seeking only the comma.

Specifying a Drive and Path

You can store library files in the current directory, in another directory on the same drive, or in another directory on a different drive.

For example, to direct the output library file for the MYTAGS.GML file to a directory named TAGS, issue the following command:

```
dtlc mytags, \tags\
```

To perform the same operation with a different library name, issue the command like this:

```
dtlc mytags, \tags\alltags
```

This directs the formatted elements to the ALLTAGS.DTL and ALLTAGS.HLP libraries located in the TAGS directory.

If the TAGS directory resides on a different drive (the D drive, for example), you would issue this command:

```
dtlc mytags, d:\tags\alltags
```

Replacing and Removing Existing Library Elements

If the output DTL library contains an existing element with the same name as an element of the same type being compiled, the compiler marks the element as deleted from the library and adds the new element. To prevent this problem, use the /PRESERVE option of the DTLC command. This option specifies that the new element will not replace an existing element with the same name.

Similarly, the /PRESERVE option prevents a single HLP library from being completely overwritten.

This DTLC command compiles and directs the output of the MYTAGS.GML file to the ALLTAGS.DTL and ALLTAGS.HLP libraries, and specifies the /PRESERVE option using the minimum abbreviation:

```
dtlc mytags, alltags /p
```

If there is an existing element in the ALLTAGS.DTL library that shares the same name as an element of the same type in the MYTAGS.GML file, the compiler issues a warning message and does not replace the existing element. If there is an existing ALLTAGS.HLP library, the compiler issues a warning message and the library will not be replaced.

When the compiler deletes an existing element within a DTL library, the element continues occupying storage in the library, even though it cannot be accessed. To free this unused storage, use the DTLC /COMPRESS option. We recommend that you use the /COMPRESS option frequently to conserve space in DTL libraries.

Note: You cannot use the /COMPRESS option for compressing HLP libraries.

This command directs the dialog elements in MYTAGS.GML to the ALLTAGS library and deletes and removes from the DTL library any previously deleted elements or any existing elements with identical name values.

```
dtlc mytags, alltags /com
```

Deleting Dialog Elements

You can use OS/2 commands to delete library files. If you want to delete those non-help dialog elements that are no longer needed and are not replaced by elements having the same name, you can use the DTLD command. Without the DTLD command, non-help dialog elements with obsolete names would continue to occupy space within a DTL library.

When you specify dialog elements with the DTLD command, the elements are deleted from the library list. The elements remain in the storage space within the library unless you specify the /COMPRESS option.

You identify the dialog element to be deleted by the value assigned to the NAME attribute of the element, or, in the case of command tables, the application identifier appended to the string CMDS. Because different types of elements may share the same NAME value, you need to specify the type of element to be deleted. The types of elements you can specify for deletion are:

- CMDTBL (command table)
- ICON (icon resources)
- KEYL (key mapping lists)
- MSGMBR (message members)
- PANEL (application panels).

You can only specify one of the previous types of elements in each DTLD command call. For example, to delete the application panel "ordpan01" from the MYTAGS.DTL library, issue this DTLD command:

```
dtld mytags, panel, ordpan01
```

DTLD assumes the library extension .DTL. The element type and element name are separated by commas in the command.

You can also specify multiple elements to be removed in a single DTLD command call.

However, because you can only specify one element type in a DTLD command call, all the elements must be of the same type. Each element must be separated by blanks or plus signs.

For example, this command removes the key mapping lists "key55," "key56" and "key88."

```
dtld mytags, key1, key55 key56 key88
```

You could also issue this DTLD command like this:

```
dtld mytags, key1, key55 + key56 + key88
```

This command removes a command table with the application identifier XYZ from the ABCCMDS.DTL library.

```
dtld abccmds ,cmdtbl ,xyzcmds
```

The DTL Display Utility

If you want to see your formatted application panels and messages before you write the Dialog Manager application code for them, can use the display utility to display them. The application panels and messages you view must be in a .DTL file (a .GML file that has been compiled by the DTL compiler).

Note: You cannot use the display utility to view the help panels you create for your application.

Accessing the Display Utility

To use the display utility, enter the following command at the OS/2 command prompt:

```
DMDISPLY
```

Following the LOGO panel, the display utility main panel appears.

Displaying Panels and Messages

There are two ways to access libraries from the main panel: the **Library Name** field or the **File** action bar choice.

Library Name field
Enter, in the **Library Name** field, the name of the library you want to access. You must specify the full path, and the .DTL extension. You do not have to specify the path if the .DTL file has been added to your DPATH.

When you enter the library name, a list of the *message-identifiers* in the library is displayed in the **Message List** list box. A list of the application panel NAME values in the library is displayed in the **Panel List** list box. Select a value from either one of these list boxes to display the formatted element.

File action bar choice
Select the **File** action bar choice to display the associated pull-down. Select the **Open** pull-down choice.

The current path name is displayed in the **Directory is:** field, and the default output "*.DTL" is displayed in the **File name:** field.

Two list boxes are also shown in the "Open" panel: **Files**, which displays a list of libraries residing on the current path, and **Directories**, which displays a list of current paths and directories. From either list box, you can access the library that contains the elements you want to display.

- If the library that contains the element you want is in the current path, you can select the library name from the **Files** list box. The display utility main panel reappears, showing a list of the *message-identifiers* in the library displayed in the **Message List** list box. It also shows a list of the application panel NAME values in the library displayed in the **Panel List** list box. Selecting a value from either one of these list boxes will display the formatted element for the value.
- If the library that contains the element resides on a different directory, you can select the directory from the **Directories** list box. The **Directory is:** field is updated with the directory you selected. The **Files** list box is updated with the list of libraries in the current path. Selecting a library name in the **Files** list box causes the display utility main panel to reappear. Selecting a value from either one of these list boxes will display the formatted element for the value.

Messages are displayed in a Dialog Manager message panel. Application panels are displayed in windows.

You can exit from any displayed message or panel using the system icon **Close** pull-down choice.

Restrictions on the Display Utility

There are two types of panels that the display utility cannot process and, therefore, cannot display. These types include help panels and panels containing user controls.

Generating the Executable File

The following section contains information on compiling the Dialog Manager source code and the linking process using the OS/2 Linker (LINK). Building a Dialog Manager application requires creating source code using programming techniques recommended in *Dialog Tag Language Guide and Reference*, and *Dialog Manager Guide and Reference*.

The DMDEMO.MAK sample program demonstrates the build process using the C/2 compiler. The DMDEMO.MAK source files are located in the \TOOLKT12\C\SAMPLES\DM\DMDEMO subdirectory. The DMDECOBL.CMD sample program demonstrates the build process using the COBOL/2 compiler. The DMDECOBL.CMD source files are located in the \TOOLKT12\COBOL\SAMPLES\DM\DMDEMO subdirectory.

To build the application's executable file:

- Compile or assemble the source code using the appropriate language compiler. Object (.OBJ) files are the output.
- Write the module definition (.DEF) file.
- Link the object files with the module definition file and any library (.LIB) files. The executable (.EXE) file is the output.

Compiling Language Specific Source Code

For each of the supported languages, the Dialog Manager provides a special include file to include in the application program's code. If you include this file, your application will automatically contain the Dialog Manager procedure declarations and a declaration of the Dialog Manager communication area. If you do not include this file, you will need to add this information to your application code.

C/2

Dialog Manager provides the special include file called ISPCAST.H for C/2 applications. This include file takes care of the typecasting necessary for use by small and medium program models. The include file also provides a type declaration for the DMCOMMBLOCK parameter, which specifies the Dialog Manager communication area structure.

If written in C/2, user control source files must be compiled with a customized storage model described in the C/2 *Compile, Link, and Run* manual. One of the following must be specified:

- The /Azzu option
- The /Azzw option, and the "_loadds" keyword added to the control procedure function.

Note: "zz" specifies the compilation model. Refer to "Creating Customized Storage Models" in the C/2 *Compile, Link, and Run* manual.

In addition, when you compile user control source files, you must specify the −GS option to disable stack checking.

Note that many of the C/2 library functions are not reentrant so special precautions may be necessary to use them within a user control. Refer to the C/2 *Compile, Link, and Run* manual for more details.

COBOL/2

Dialog Manager provides the special include file called ISPCOBOL.INC for COBOL/2 applications. The /LITLINK option must be specified.

Dialog Manager is not supported for COBOL small or compact program models; therefore, do not use the /MODEL"COMPACT" or /MODEL"SMALL" options when compiling.

FORTRAN/2

Dialog Manager provides the special include file called ISPFORT.INC for FORTRAN applications. Dialog Manager does not require any special commands when compiling a FORTRAN/2 application.

Macro Assembler/2

Dialog Manager provides the special include file ISPCALL.INC for MASM applications. To use this file in your application, specify

```
INCLUDE ISPCALL.INC
```

This file provides a struc definition of the Dialog Manager communication area and ensures that the far calls are made to the Dialog Manager routines.

Pascal Compiler/2

Dialog Manager provides the special include file called ISPPASC.INP for Pascal applications. This file provides a TYPE definition for the Dialog Manager communication area and supplies definitions for the Dialog Manager ISPCI and ISPCI2 service calls.

Writing the Module Definition File

The module definition (.DEF) file is an ASCII text file that can be created with any suitable text editor. An example of a module definition file is provided in the \TOOLKT12\C\SAMPLES\DM\DMDEMO subdirectory and is shown in the following sample.

```
NAME     dmdemo  WINDOWAPI

DESCRIPTION 'Sample Application'

STUB     'OS2STUB.EXE'

CODE     MOVEABLE
DATA     MOVEABLE MULTIPLE
```

The module definition file statements tell the OS/2 linker (LINK) about the executable file it is creating. Following are explanations of the statements used in the DMDEMO.DEF sample program:

NAME
tells LINK the executable file being created is a program module called DMDEMO. WINDOWAPI indicates that this program must execute as a Presentation Manager windowing application. This statement is mandatory.

STUB
sets up a stub file that generates an error message if the user attempts to execute a Presentation Manager application in the DOS environment. OS2STUB.EXE is a DOS executable file that is added to the beginning of the program module.

CODE
defines the attributes of all code segments in DMDEMO.

DATA
defines the attributes of all data segments in DMDEMO.

Linking the Object Files

To generate your application's executable file, use the LINK utility. You need the following files:

Object Files
The object (.OBJ) files are created by your language compiler or assembler. You must compile or assemble all application source code files.

Library Files
LINK resolves intersegment calls using library (.LIB) files. If you have called external subroutines or subroutines in dynamic link libraries, you must link the library files together with the object files. You can specify a library search path using the SET LIB=path command before linking.

Module Definition File
This file tells LINK about the executable file it is creating.

Other Linking Considerations

When linking a Dialog Manager application, you must use the /STACK option to allow enough stack space for the Dialog Manager to operate. A minimum starting value of /STACK:20000 is recommended. This value applies to all languages, and may increase or decrease based on the application's size and complexity.

You can specify the /STACK option on any LINK command or place the /STACK option in a module definitions (.DEF) file. For information about using the LINK command, consult the compile, link, and run documentation for the language you are using.

A Dialog Manager application is also a Presentation Manager application. Therefore, a Dialog Manager application needs to run in a Presentation Manager window. To enable your Dialog Manager application to run in a windowing environment, you need to link with a module

definitions (.DEF) file, which contains the statement:

```
NAME modulename WINDOWAPI
```

where *modulename* is the name of your module or application. The WINDOWAPI attribute is required.

C/2, FORTRAN/2, Macro Assembler/2, Pascal Compiler/2

The libraries language links to are:

Language	Library
C	C_ISP.LIB
FORTRAN	FO_ISP.LIB
MASM, Pascal	P_ISP.LIB

COBOL

For COBOL applications, the library to link to depends on the program model being used.

The specific library or object module to use with each program model is given by the following chart:

COBOL Model	Library
Huge	CO_ISP.LIB
Large	CO_ISP.LIB
Medium	CO_ISPM.LIB

When linking programs compiled with the COBOL/2 compiler for medium program models, you must specify the /DO option.

REXX Programming Aid

In REXX, a Dialog Manager application is considered to be a Presentation Manager program. Therefore, a Dialog Manager application needs to run in a Presentation Manager window. To enable your Dialog Manager application to run in a windowing environment, you need to initiate the application with the following statement:

```
PMREXX programname
```

where *programname* is the name of your application.

Chapter 4. Building Text Window and Full-Screen Applications

This chapter describes how to build a text window, full-screen OS/2, or full-screen DOS executable file. The chapter also describes how to build a family application that can execute in the OS/2 and DOS environments of OS/2 and in the DOS 4.0 environment. The chapter includes compiling the source code, linking the object code using the OS/2 Linker (LINK), binding a family executable file using the BIND utility, and building a text message file using the MKMSGF and MSGBIND utility.

To demonstrate the build processes, we show the VIOSAMPC.C sample program located in the \TOOLKT12\C\SAMPLES\BSE\VIOCBAT subdirectory. To illustrate how to create and display a text message, we refer to the WTC.C sample program in the \TOOLKT12\C\SAMPLES\BSE\WTCBAT subdirectory.

The Build Processes

The build processes described in this chapter assume you have written your source code following programming techniques recommended in the *Programming Guide*. Application Programming Interface (API) calls used in your source code must be compatible with the type of application you are writing.

For example, API calls used in family applications must be part of the subset described in the *Programming Guide* and are labeled FAPI. The VIOSAMPC.C sample program contains such calls and can be linked and bound to run in both the OS/2 and DOS environments.

API calls used in full-screen OS/2 and DOS applications must be compatible with those execution environments. Text window applications must contain calls that are compatible with the Presentation Manager.

Figure 4-1 illustrates the build process for text window and full-screen OS/2 applications. The additional tasks needed to bind a family application to run in both the OS/2 and DOS environments are illustrated below the dotted line.

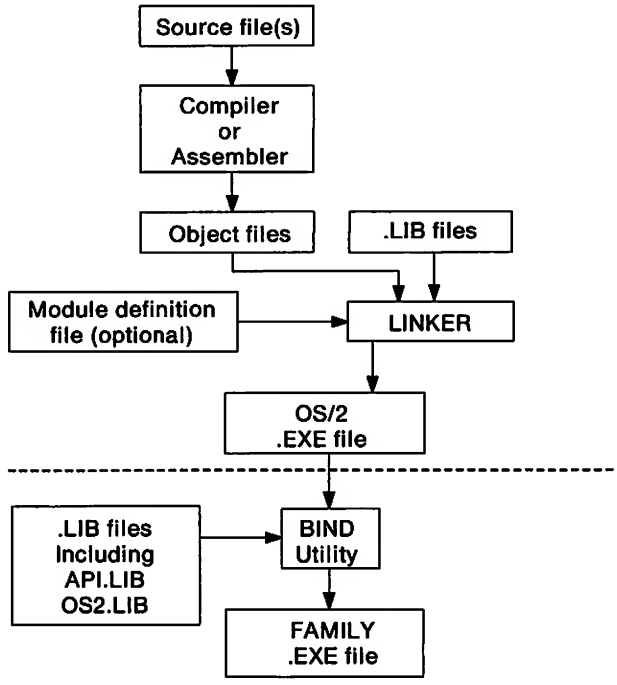

Figure 4-1. Building a Text Window, Full-Screen, or Family Application

To build the application's executable file:

- Compile or assemble the source code using the C/2 compiler or Macro Assembler/2. Object (.OBJ) files are the output.
- Write a module definition (.DEF) file, if required. The module definition file is required if you are building a text window application. Otherwise, it is optional.
- Link the object files with the (optional) module definition file and any library (.LIB) files. The executable (.EXE) file is the output.
- If you are building a family application, bind the executable file to execute in the DOS environment.

Compiling or Assembling the Source Code

Compile or assemble the source code to generate object (.OBJ) files. See your language compiler or assembler manual for specific information. The VIOSAMPC OS/2 executable file is built using the

VIOCBAT.CMD in the \TOOLKT12\C\SAMPLES\BSE\VIOCBAT subdirectory. You can compile the VIOSAMPC.C source code using the following command, which is case sensitive:

```
cc viosampc.c,,,/Fc /Zl /Zp /Ze /G0 /Gs /AM;
```

where:

/Fc generates a default Code (.COD) file containing the assembler language for each line of code in C. The file is useful for line-by-line debugging.

/Zl specifies not to use default libraries.

/Zp specifies that fields in records should be packed to conserve memory; OS/2 API calls expect packed fields.

/Ze specifies that the predefined keywords, the identifiers used by the C compiler, should be enabled. This option is not necessary unless C was installed to be compatible with Systems Application Architecture (SAA).

/G0 specifies that 8086/8088 instructions should be generated.

/Gs specifies that stack probes should not be used; use this option only if the application has been completely debugged.

/AM specifies a medium memory model.

The first comma in the command delimits the list of source files. In this case, VIOSAMPC.C is the only source file. The second comma tells the compiler to create an object file called VIOSAMPC.OBJ using the default source file name and file extension. The third comma tells the compiler to generate a *list file* called VIOSAMPC.LST which, again, uses the default file name and extension. The semicolon at the end of the command ends the command.

Writing the Module Definition File

A module definition (.DEF) file is a text file that can be written using any ASCII text editor. The module definition file is optional for full-screen OS/2 and DOS applications and family applications. The statements and keywords in the module definition file provide LINK with information about the executable file it is creating, such as defining the attributes of code and data segments, imported functions, and heap and stack sizes. Many of these statements have appropriate defaults, so you need not declare them. For a complete list of the statements, see Chapter 8, "Module Definition File Statements."

The module definition file is required for text window applications. An example of a module definition file for a text window application is illustrated in Figure 4-2.

```
NAME    Viosampc  WINDOWCOMPAT

DESCRIPTION 'OS/2 Text Window Sample Application'

STUB  OS2STUB.EXE
```

Figure 4-2. Module Definition File for a Text Window Application

where:

NAME
: tells LINK the executable file being created is called VIOSAMPC. WINDOWCOMPAT sets a flag in the executable file's header, meaning that this application can execute in a default window supplied by OS/2. This statement is mandatory if the program is to run in a window environment.

STUB
: sets up a stub file that generates an error message if the application user attempts to execute the application in the DOS environment. The OS2STUB.EXE file is provided in the \TOOLKT12\BIN subdirectory.

Linking the Object Files

To generate your application's executable file, use the LINK utility. You need the following files:

object files
: the output of the language compiler or assembler

library files
: the code to resolve external references

module definition file (optional)
: provides LINK with information about the executable file it is creating.

The VIOSAMPC executable file is generated by linking the output of the compiler, the VIOSAMPC.OBJ file, together with the following three library files, which are provided with *IBM C/2, Version 1.1* and the tools package:

MLIBCE.LIB
: the combined library for the C medium run-time library. Note that when the C/2 compiler is installed, you can request a single, combined library for each memory model. In this case, the medium memory model was selected for an application comprised of many code segments and one 64KB data segment.

OS2.LIB
: the resolutions for the OS/2 API calls in dynamic link libraries.

Note: Another library file, DOSCALLS.LIB, can be used to recompile or relink applications that have not been modified to run in the OS/2 Version 1.1 environment. If the application was modified, we recommend that it be recompiled and relinked using OS2.LIB in the \TOOLKT12\LIB subdirectory.

To link the object and library files together with the sample module definition file, use the following command:

```
link viosampc,,,mlibce.lib os2.lib;
```

The first comma in the command delimits the list of object files. In this case, VIOSAMPC.OBJ is the only object file. The second comma tells LINK to create an executable file called VIOSAMPC.EXE using the default executable file name and file extension. The third comma tells the LINK to generate a *map file* called VIOSAMPC.MAP which, again, uses the default file name and extension. See Chapter 6, "The LINK Utility" for more information about map files. The semicolon at the end of the command delimits the list of library files and tells LINK that there is no module definition file.

The OS/2 executable file VIOSAMPC.EXE is ready to run.

Binding the Application to Run in the DOS Environment

The BIND utility creates a family executable file from an OS/2 executable file. The family application can execute in the DOS 3.3 environment, the OS/2 environment, and the DOS environment of OS/2. The family API calls that execute in the DOS environment are contained in the API.LIB file in the \TOOLKT12\LIB subdirectory. The entries to resolve the OS/2 API calls are contained in the OS2.LIB file in the \TOOLKT12\LIB subdirectory.

Note that the BIND utility is designed to be used with the IBM Linker/2 Version 1.1. To BIND an OS/2 executable file for the DOS environment, type the following command:

```
BIND executable [libraries]
[objectfiles]
[/o executable_dest]
[/m [mapfile]]
[/n [@]name]
```

where:

executable
: specifies the name of the OS/2 executable file being bound. The executable file name can include a drive and directory path.

libraries
: specifies the names of the libraries to be searched to resolve external references. You can specify more than one library name.

objectfiles
: specifies the names of the object (.OBJ) files to be bound together.

/o executable_dest
: specifies the name, drive, and path of the DOS executable file. If not specified, the bound executable file overwrites the OS/2 executable file.

/m mapfile
: specifies the name and directory path of the map file for the DOS environment. If not specified, the default is the executable file name with the file extension .BM.

/n @name
: specifies the individual names of subroutines or the name of the file containing a list of all the subroutines. If you do not have many routines, you should use only the /n option.

The VIOSAMPC.EXE file can be bound to run in the DOS environment using the following command:

```
BIND viosampc.exe
\toolkit12\lib\os2.lib
\toolkit12\lib\api.lib
/o fapisamp.exe
```

Building a Text Message File

Text-based messages to display error, help, prompt, or general information to the application user can be created using the MKMSGF utility in the tools package. Text messages in text window and full-screen applications do not need to be loaded into memory with the application but can reside on disk until needed. Or the messages can be bound to the executable file using the MSGBIND utility. Binding the message to the executable file ensures that the message is always accessible. One message file can serve more than one application. The message file can be translated easily into other national languages.

Note that these text message facilities are not used in the Presentation Manager environment. Text and graphic messages in Presentation Manager applications are provided by way of message boxes.

Figure 4-3 illustrates a sample text message file taken from the MESSAG.TXT file in the \TOOLKT12\C\SAMPLES\BSE\WTCBAT subdirectory:

```
MES
MES0000I: (mm%4dd%4yy) %2%4%1%4%3
MES0001I: (dd%4mm%4yy) %1%4%2%4%3
MES0002I: (yy%4mm%4dd) %3%4%2%4%1
MES0003I: Current Date is: %0
MES0004I: La data attuale e: %0
```

Figure 4-3. Example of a Text Message Source File

Where:

MES0000I - MES0004I
: identifies message numbers in sequential order. The first three characters indicate the component identifier; the four-digit number indicates the message number, which is followed by a colon and a blank space. You must supply a component identifier, such as prn for a printer or dsk for a disk. If a message number is not used, type the number and leave an empty entry, for example:

```
MES00005I:
MES00006I:  Message 5 was not used.
```

I
: indicates the message contains information (I) for the application user. Other categories include help (H), error (E), warning (W) and prompt (P).

%0
: displays a prompt for input from the user, after which a carriage return and line feed are inserted.

Current date is:
: contains the text of the message displayed to the user, followed by a prompt for input; the text message can be more than one line.

La data attuale e:
translation of the message displayed to the user, followed by a prompt for input.

Figure 4-4 illustrates how to convert the source text file to a binary file using the MKMSGF utility.

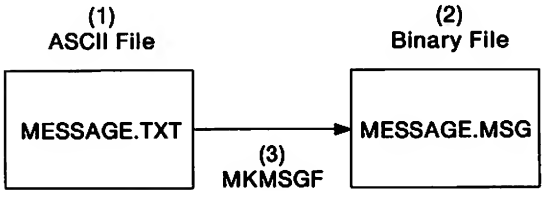

Figure 4-4. Converting the Text File to Binary Format

In Figure 4-4 MESSAG.TXT is the source text file, and the output is a file called MESSAG.MSG. The default file extension is .MSG. If you change the text of a message in your source file, you must run the MKMSGF utility again to convert it to a binary format. To start MKMSGF, type the following command:

```
MKMSGF [infile] [outfile]
```

Name the outfile using the three character component identifier and the .MSG file extension, for example: MES.MSG.

Help Messages

You can create a corresponding help message file which the application user can display using the HELP command. Help can be displayed for warning and error messages. The help message must be an ASCII text file separate from the message file. The category must be H. Otherwise, the help message file is built in the same manner as a message file.

The number used for each help message must be the same as the number used for the error or warning message in the message file. When you run the MKMSGF utility on help message files, you must name the outfile using the three-character component identifier followed by the letter H and the .MSG extension, for example: MESH.MSG.

Displaying the Message

Figure 4-5 illustrates how to display a message to the user of the application. The code is excerpted from the WTC.C sample program in the \TOOLKT12\C\SAMPLES\BSE\WTCBAT subdirectory.

```
/*****************************************************************************
 * The DosGetMessage function retrieves a message from the specified system
 * message file and inserts variable information into the body of the
 * message.  This call gets the 'Current Date is: ' message.
 *
 *         Parameters: ivtable         Table of variables to insert.
 *                     NO_INSERT       Number of variables to insert.
 *                     dataarea        Buffer address to return message.
 *                     DATALEN         Length of the buffer.
 *                     leader          Number of the message.
 *                     msgfile         Message file path name.
 *                     msglen          Length of the returned message.
 *
 *****************************************************************************/

   error_code = DosGetMessage((PCHAR FAR *)ivtable, NO_INSERT,
                    (PCHAR)dataarea, DATALEN, leader, (PSZ)msgfile,
                    (PUSHORT)&msglen);
   if (error_code != NO_ERROR) {
      printf("DosGetMessage error code= %d\n",error_code);
   }

/*****************************************************************************
 *
 * The DosPutMessage function outputs the message in a buffer passed by a
 * caller to the specified handle.  The function formats the buffer to
 * prevent words from wrapping if displayed to a screen.  This call puts
 * the 'Current Date is: ' message on the screen.
 *
 *         Parameters: FILE_HANDLE     Handle of output file/device.
 *                     msglen          Length of the message.
 *                     dataarea        Message buffer.
 *
 *****************************************************************************/

   error_code = DosPutMessage(FILE_HANDLE, msglen, (PCHAR)dataarea);
   if (error_code != NO_ERROR) {
      printf("DosPutMessage error code= %d\n",error_code);
   }
```

Figure 4-5 (Part 1 of 2). Displaying the Message to the User

```
/******* This section of code determines the output format of the date ******/

    switch (date_format) {

    case 0:  printf("%d%s%d%s%d\n", dmonth, cinfo.szDateSeparator,
                   dday, cinfo.szDateSeparator, dyear);
             break;
    case 1:  printf("%d%s%d%s%d\n", dday, cinfo.szDateSeparator,
                   dmonth, cinfo.szDateSeparator, dyear);
             break;
    case 2:  printf("%d%s%d%s%d\n", dyear, cinfo.szDateSeparator,
                   dmonth, cinfo.szDateSeparator, dday);
             break;
    default: printf("error in COUNTRY.SYS file %d\n",cinfo.fsDateFmt);
             break;
    } /* end case */

    ccode.country = ITA_CODE;           /* set country code for Italy    */
    leader  = ITA_MSG;            /* set pointer to Italian message in file   */

    } /* end while */

/*****************************************************************************
```

Figure 4-5 (Part 2 of 2). Displaying the Message to the User

Note that the variables are declared at the beginning of the file and are not excerpted here. To run the WTC sample program, see the WTCBAT.C batch file which contains the compiler and link options necessary to execute the application. Although not illustrated here, the API call DosInsMessage can be used to insert variable text strings into the message. Use this call if the message is to be loaded before the content of the message is known.

Binding a Message to the Executable File

When the DosGetMessage API call is issued, it first searches for the message in bound data segments, if any, and then in the message files. To ensure that a message is displayed quickly, you can bind it to the application's executable file using the MSGBIND utility. Type the following command:

```
MSGBIND infile
```

An example of an infile follows:

```
>PROG1.EXE
<\MESSAGES\PRGMSG.MSG
PRG0100
PRG0101
PRG0102
<\MESSAGES\APP.MSG
APP0001
APP0002
APP0003
```

Where:

>PROG1.EXE
: is the executable file to be modified

<
: defines the first message of a series to be bound, delimited either by the end of the series or a greater-than symbol (>).

<\MESSAGES\PRGMSG.MSG and <\MESSAGES\APP.MSG
: names the files containing the binary versions of the messages (created by MKMSGF) and their identifying numbers: the three-character component identifier and the four-digit message number.

Chapter 5. Building a Dynamic Link Library

This chapter describes how to build a dynamic link library. The chapter includes compiling, using the OS/2 Linker (LINK), and creating an import library. We demonstrate the build process with the TYPETEXT sample programin the tools package. The TYPETEXT source files are located in the \TOOLKT12\C\SAMPLES\PM\TYPETEXT subdirectory.

Common subroutines that can be used by more than one application can be placed in a dynamic link library. A subroutine in a dynamic link library has the same format and structure as an executable file. An application is written as though the subroutine is in a different code segment and references to the subroutine are made with a standard intersegment (far) call.

At compile time, the compiler (or assembler) generates an external reference point in the object files. Along with the dynamic link library object files and optional library files, a module definition file is mandatory input to LINK. The module definition file tells LINK that the library it is creating is a dynamic link library and defines the subroutines that are available for *export*.

To complement the dynamic linking services, segments in a dynamic link library can be designated to be loaded only if the application explicitly requests them at run-time. Because the format and structure of a dynamic link library are the same as for an executable file, individual segments can be defined in the module definition file as loadable *on demand*. This means that the segments are not brought into memory unless the application explicitly requests that they run. This facility is particularly suitable for error subroutines.

The Build Process

Figure 5-1 illustrates the build process.

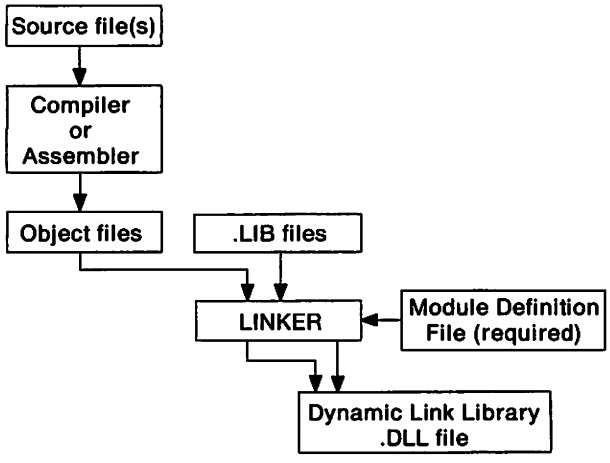

Figure 5-1. Building a Dynamic Link Library

To build a dynamic link library:

- Write the source code for the dynamic link library file.
- Compile or assemble the source code using the C/2 compiler or Macro Assembler/2. Object (.OBJ) files are the output.
- Write the module definition (.DEF) file.
- Link the object files with the module definition file and any optional library files. The dynamic link library (.DLL) file is the output.

Writing the Source Code

Dynamic link libraries can be written in any of the languages supported by OS/2, but levels of support vary, and restrictions apply. Only C/2 Version 1.1 and Macro Assembler/2 Version 1.0 provide support for dynamic link libraries. COBOL/2 Version 1.0 supports only one entry point in each dynamic link library. Refer to your language manual for additional information.

The subroutine TypetextSetColor from the TYPETEXT sample program is used to demonstrate the process for building program subroutines in a dynamic link library. The TypetextSetColor source code is moved into a separate file called TYPEDLL.C and compiled. The object file is then built in a dynamic link library.

Moving the subroutine out of the main program requires some additional source code and a header file declaring the subroutine in the dynamic link library. This header file must be included in the application that uses the subroutine. In this example TYPEDLL.H would be added to TYPEDLL.C and to TYPETEXT.C.

The following is the only subroutine declaration in TYPEDLL.H:

```
VOID EXPENTRY TypetextSetColor (HPS);
```

The EXPENTRY statement is required to ensure the subroutine is called using the Pascal far calling convention. If you omit this statement, the IMPLIB utility will not generate the correct library (.LIB) file.

All changes and additions to TYPEDLL.C are highlighted in **bold** print. The source code for TypetextSetColor is illustrated in Figure 5-2.

```
#define INCL_PM
#include <os2.h>
#include "typedll.h"

int _acrtused=1;

VOID EXPENTRY TypetextSetColor(hps)
HPS hps;
{
    LONG  WindowTextColor;
    LONG  WindowColor;
    CHARBUNDLE cb;

    WindowColor = WinQuerySysColor(HWND_DESKTOP,
                                   SYSCLR_WINDOW,
                                   (ULONG)NULL);

    WindowTextColor = WinQuerySysColor(HWND_DESKTOP,
                                       SYSCLR_WINDOWTEXT,
                                       (ULONG)NULL);
    cb.usBackMixMode = BM_OVERPAINT;
    cb.lBackColor = GpiQueryColorIndex(hps,
                                       (ULONG)NULL,
                                       WindowColor);
    cb.lColor = GpiQueryColorIndex(hps,
                                   (ULONG)NULL,
                                   WindowTextColor);
    GpiSetAttrs(hps,
                PRIM_CHAR,
                CBB_COLOR | CBB_BACK_COLOR | CBB_BACK_MIX_MODE,
                0L,
                (PBUNDLE)&cb);
}
```

Figure 5-2. TYPEDLL.C Subroutine

Compiling or Assembling Your Source Code

Compile the source code to generate object (.OBJ) files. If you are using C, there are restrictions for compiling dynamic link libraries. Refer to your language manuals for more information.

To compile the subroutine TYPEDLL.C to generate the TYPEDLL.OBJ file, type the following command:

```
cl /c /W2 /Alfu /Gs typedll.c
```

where:

/c specifies compile only, link is a separate step.

/W2 requests level 2 warning messages.

/Alfu ensures that all calls to other functions are far calls, all data is far, and that the data segment register should be reloaded at every function invocation. Note that the large memory model (LLIBCDLL.LIB) must be used when creating dynamic link libraries.

/Gs turns off stack checking.

Writing the Module Definition File

A module definition (.DEF) file must be used to create a dynamic link library. The module definition file is an ASCII text file and can be created with any text editor. Name this module definition file NEWDLL.DEF.

Figure 5-3 illustrates the module definition file to be linked together with the TYPEDLL.OBJ file.

```
LIBRARY

DESCRIPTION  'Sample .DEF file for
                    Dynamic Link Module'

STUB 'OS2STUB.EXE'

CODE LOADONCALL

EXPORTS
     TYPETEXTSETCOLOR @1
```

Figure 5-3. Module Definition File for a Dynamic Link Library

The module definition file statements tell LINK about the dynamic link file it is creating.

where:

LIBRARY
: tells LINK to create a dynamic link library. This statement is mandatory.

STUB
: adds the file OS2STUB.EXE, included in the tools package, to the dynamic link library (.DLL) file.

CODE LOADONCALL
: tells LINK to load all the code segments on demand.

EXPORTS
: specifies that TypetextSetColor subroutine is the first entry point in the dynamic link library. You must do this or the subroutines in the dynamic link library are not accessible to other applications.

Creating the Dynamic Link Library using the LINK utility

Creating a dynamic link library is no different than creating an executable file from object files. LINK combines your object files and optional library files to create a dynamic link library (.DLL) file .

To create your dynamic link library, you need the following files:

Object Files
: The object (.OBJ) files are created by your language compiler or assembler.

Library Files
: LINK resolves intersegment calls using library (.LIB) files. If you have called external subroutines or subroutines in dynamic link libraries, you must link the library (.LIB) files together with the object (.OBJ) files. You can specify a library search path using the SET LIB=*path* command before linking.

Module Definition File
: This file tells LINK about the dynamic link library file it is creating:

To start the LINK utility type:

```
LINK
```

at the OS/2 command prompt and press the Enter key. Type the following commands in response to the LINK prompts:

```
typedll.obj        /A:16
newdll.dll
newdll.map         /NOD
llibcdll.lib+
os2.lib
newdll.def
```

Or you can put the commands into a response file. For a description and an example of operating the LINK utility using a response file, see "Creating a Response File" on page 6-3.

The responses tell LINK to generate the dynamic link library named NEWDLL.DLL from the object file called TYPEDLL.OBJ and create a map file called NEWDLL.MAP. LINK sequentially searches the library files LLIBCDLL.LIB and OS2.LIB to resolve intersegment references and uses the module definition file called NEWDLL.DEF.

The /A:16 LINK option specifies alignment 16, which is recommended for Presentation Manager applications.

/NOD tells LINK to ignore all default libraries.

Before another application can access a subroutine in your dynamic link library, the dynamic link library (.DLL) file must be in a directory specified in the library search path (LIBPATH statement in CONFIG.SYS).

Building the Import Library

Another application can access the dynamic link library by linking with a module definition file containing the subroutines it needs to *import*. Or, you can create an *import library*.

An import library resolves the application's external references by supplying pointers to the subroutines in a dynamic link library. Using an import library saves time if there are many subroutines that need to be imported. Instead of itemizing each subroutine that needs to be imported in the module definition file, an import library generates them automatically.

The import library is created using the IMPLIB utility in the tools package. IMPLIB takes the module definition file that you used to create the dynamic link library and generates an import library file as output.

Figure 5-4 illustrates the build process.

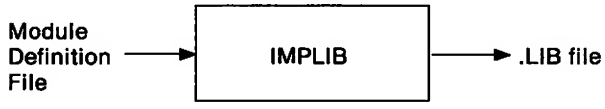

Figure 5-4. Creating an Import Library using IMPLIB

To create an import library named NEWDLL.LIB for the TYPEDLL.C file, you type the following command:

```
implib newdll.lib newdll.def
```

The library (.LIB) files must be copied to a directory in the library search path. You can specify a library search path using the SET LIB=*path* command.

Starting the IMPLIB utility

To start the IMPLIB utility, type the following command:

```
implib   lib_name  mod_def  [ mod_def ]
```

where:

lib_name
: is the name of the import library to be created.

mod_def
: is the name of the module definition file that must be input to IMPLIB. If you have more than one module definition file, separate them with blank spaces.

The error messages produced by IMPLIB are listed in Appendix A, "Error Messages."

Rebuilding the Sample Application

The following steps show you how to rebuild the TYPETEXT sample application using the TypetextSetColor subroutine in the dynamic link library:

- In the TYPETEXT.H file, remove the declaration for the TypetextSetColor subroutine.
- In the TYPETEXT.C file:

  Add the #include "typedll.h" statement following the #include "typetext.h" statement,

  Remove the TypetextSetColor subroutine.
- In the TYPETEXT.L response file, add newdll.lib:

```
typetext.obj     /A:16
typetext.exe
typetext.map     /NOD
slibce.lib+
newdll.lib+
os2.lib
typetext.def
```

- Ensure files and libraries are in the correct directories. To use the default tools directories:

  Copy NEWDLL.DLL to the directory \OS2\DLL.

  Copy NEWDLL.LIB to the library directory \TOOLKT12\LIB.
- Rebuild TYPETEXT using the TYPETEXT.CMD file located in \TOOLKT12\C\SAMPLES\PM\TYPETEXT subdirectory.

Dynamic Link Libraries and 8.3 File Names

In order to be compatible with all OS/2 file systems, dynamic link libraries must not create internal temporary files or directories that do not comply with 8.3 DOS file naming conventions.

Applications written for OS/2 Version 1.0 and Version 1.1 do not support non 8.3 file names. In order to be compatible with these applications, OS/2 Version 1.0 and OS/2 Version 1.1 dynamic link libraries may not be modified in OS/2 Version 1.2 to return non 8.3 file names to the application. For example, if a dynamic link library semaphore package returned unique semaphores named on the basis of the calling module, and *excel* was the caller, *excel.uniqueid* is not an acceptable name to return. Names passed to the dynamic link library from the application have already been filtered and so are acceptable. Dynamic link libraries new to OS/2 Version 1.2 are also subject to this restriction because the caller could be running on an 8.3 only file system and use the returned name to create a file. See the *Programming Guide* for a definitive description of long file name support.

Presentation Drivers and Queue Processors

A dynamic link library containing a Presentation driver must have a file extension of .DRV and not .DLL.

If the dynamic link library contains a queue processor then the file extension must be a .QPR.

Create the dynamic link library as described earlier and rename it when copying it to the correct directory.

Note: For a display Presentation driver, the user will rename the device driver .DRV file to DISPLAY.DLL.

For additional information and examples on Presentation drivers and queue processors, see *I/O Subsystems and Device Support Volume 2: Presentation Driver Interface*.

Chapter 6. The LINK Utility

The OS/2 Linker (LINK) utility, provided with the OS/2 product, translates your application object code to executable code. LINK can be used to produce executable code for all environments supported by OS/2, including Presentation Manager, Dialog Manager, text window, full-screen OS/2, and full-screen DOS. In addition, LINK generates an executable file for the family application that can execute in the OS/2 and DOS environments. LINK also generates dynamic link libraries, which are described in Chapter 5, "Building a Dynamic Link Library."

To link your application's object files with optional libraries, you can select the method that is suitable for you. You can respond to a series of LINK prompts, type commands directly at the command prompt, or create a response file that can be reused.

Responding to LINK Prompts

After you type LINK at the command prompt, a series of five prompts will appear, one at a time. They include:

```
Object modules [.OBJ]:
Run file [filename.EXE]:
List file [NUL.MAP]:
Libraries [.LIB]:
Definitions file [NUL.DEF]:
```

You can respond using any combination of upper-case and lower-case letters. Enter your responses by pressing the Enter key.

To terminate the linking process at any point, press the Ctrl + Break keys simultaneously.

LINK supplies the following default file extensions: .OBJ, .EXE, .MAP, .LIB, and .DEF. You can override these extensions by typing the file extension of your choice.

Typing a semicolon and pressing the Enter key at any point in the process selects all remaining default responses.

Specifying LINK options allows you to tell LINK how to translate your source code. All the options are described in Chapter 7, "LINK Options."

Using the LINK prompting method, you can specify options anywhere on the response line except before a comma at the end of a line of characters. If you want to specify more than one option, group them at the end of a response. Or, you can specify them at the end of several responses. Each option must begin with a slash (/).

To start LINK, type:

```
LINK
```

at the OS/2 command prompt and press the Enter key.

LINK displays the first prompt.

```
Object modules [.OBJ]:
```

At this prompt, type the object files that you want linked together in your executable file. You must enter the name of at least one object file. Include the drive and path if the files are not in the working directory. Separate the file names with plus (+) signs or blank spaces. If you type the plus sign as the last character at the end of a line and press the Enter key, the prompt will reappear, allowing you to type the names of additional object files.

LINK automatically supplies the .OBJ file extension. Unless your files have other extensions, you do not need to type the file extensions.

LINK displays the next prompt:

```
Run file [filename.EXE]:
```

At this prompt, you can specify a file name for your executable file or press the Enter key.

If you press the Enter key, LINK uses the file name displayed at the prompt and adds the file extension .EXE.

If you specify a file name, LINK adds the file extension EXE.

LINK displays the next prompt:

```
List file [NUL.MAP]:
```

At this prompt, enter the name of the list (map) file you want to create. If you do not give a file name extension, LINK uses .MAP by default. By adding

the /MAP option you can request a list of the addresses of the public symbols used in the application. If you press Enter at this prompt without specifying a file name, LINK uses the special file name NUL.MAP, which tells LINK not to create a map file.

A map file lists the names, memory addresses, and lengths of the segments in the application. If you did not compile and link your application to be debugged using a symbolic debugger, the map file is useful because it lists the errors encountered during the link process. LINK puts the map file into the working directory.

Examples of map files for a full-screen OS/2 and full-screen DOS application are shown on page 6-5. A map file is not necessary for debugging a Presentation Manager application because it is compiled and linked for debugging using a symbolic debugger.

LINK displays the next prompt:

```
Libraries [.LIB]:
```

At this prompt, type the names of the library files that you want to link to the executable file or press the Enter key to advance to the next prompt.

If you have more than one library file, separate the file names using a plus (+) sign. Include the drive and path if the files are not in the working directory. If a plus sign is the last character typed on the line, the prompt reappears, allowing you to specify more library files.

LINK automatically supplies the .LIB file extension. Unless your files have other extensions, you do not need to type the file extensions.

If your library files are not in the working directory, you can specify the order in which you want LINK to look for them. Before you begin the linking process, specify the search path using the SET LIB= command. Each search path can be either a directory specification or a library name. Directory specifications must end with a back slash (\) so that LINK can distinguish the directory names from the library names.

To locate the default libraries, LINK searches in the following order:

1. The current working directory.
2. The directories in the order listed following the *libraries* prompt.
3. The directories specified by the LIB environment variable.

If the library name includes a directory specification, LINK searches only that directory for the library. LINK searches default libraries after libraries given on the command line.

LINK displays the following prompt:

```
Definitions file [NUL.DEF]:
```

Type the name of a module definition file for the executable file or dynamic link library if required. Otherwise press the Enter key.

After these responses, LINK creates the executable file or dynamic link library. It returns you to the OS/2 command prompt or displays an error message. The LINK error messages are listed in Appendix A, "Error Messages."

Example of Responding to Prompts

The following example shows how to link the object files called MODA.OBJ, MODB.OBJ, MODC.OBJ, and STARTUP.OBJ:

```
LINK
Object modules [.OBJ]:  moda+modb+
Object modules [.OBJ]:  modc+startup
Run file [.EXE]:  moda.exe
List file [NUL.MAP]:  abc  /MAP
Libraries [.LIB]:  b:\lib\math
Definitions file [NUL.DEF]:
```

Entering a plus (+) sign as the last character on the line prompts LINK to reissue the previous prompt. LINK joins the two lines when it runs. The /MAP option tells LINK to list all the public symbols in the application. LINK searches for the library file MATH.LIB in the \LIB directory on drive B:. An executable file named MODA.EXE and a map file named ABC.MAP are created.

Typing Input as a Single Command

To begin running LINK with a single command, supply the necessary parameters as the LINK command.

An example of the form of the LINK command that should be typed at the OS/2 command prompt is:

```
LINK objlist[,runfile][,listfile]
[,libraries][,definitionsfile]
[/optionslist]
```

where:

objlist
: is a list of object files that you want linked together. The file names should be separated by plus (+) signs or blank spaces. If you do not specify file name extensions, LINK provides the default extension .OBJ.

runfile
: is the name of the executable file LINK is creating. If you do not specify a file name, LINK uses the name of the first object file. The file name is given the extension .EXE if it is an executable file, and .DLL if it is a dynamic link library.

listfile
: is the name of the file that contains the map listing. The default file name extension is .MAP.

libraries
: is a list of libraries for LINK to search. The library names should be separated by plus (+) signs or blank spaces.

definitionsfile
: is the name of the module definition file for the executable file or dynamic link library.

/optionslist
: is a list of LINK options. Options can be specified at any point in the command line.

Note: Responses within a command line are separated by commas.

Examples of using a Single Command

The following example tells LINK to link the object file FILE.OBJ and create the executable file FILE.EXE. It creates the map file called FILE.MAP and searches the library ROUTINE.LIB for routines and variables used in the application.

```
LINK file.obj,file.exe,file.map,
                              routine.lib
```

The following example tells LINK to link the object files FUN.OBJ, TEXT.OBJ, TABLE.OBJ, and CARE.OBJ. By default, the executable file created is named FUN.EXE. A list file named FUNLIST.MAP is created. LINK searches for unresolved references in the library file called COBLIB.LIB.

```
LINK FUN+TEXT+TABLE+CARE,,FUNLIST,
                                COBLIB.LIB;
```

The following example tells LINK to link the two object files called STARTUP.OBJ and FILE.OBJ (located on the current drive) to create an executable file named FILE.EXE on drive B:. LINK also creates a map file called FILE.MAP in the \MAP directory of the current drive and searches only the default libraries.

```
LINK startup+file,b:file,\map\file;
```

The following example tells LINK to link the object file SAMPLE.OBJ using the module definition file SAMPLE.DEF and the libraries LIB1.LIB and LIB2.LIB. LINK creates the executable file SAMPLE.EXE and the map file SAMPLE.MAP. It searches the library files LIB1.LIB and LIB2.LIB to resolve any external references in the SAMPLE.OBJ file.

```
LINK sample/A:4,sample.exe,
sample.map/LI, lib1+lib2 /NOD, sample
```

The /A option sets the segment alignment factor to 4. The /LI option tells LINK to copy the line number information from the object files. The /NOD option tells LINK to ignore any default libraries specified in the object file.

Creating a Response File

To operate LINK using a response file, you must first create a file that contains the responses you want LINK to process. You can give the file any name, and create it with any text editor.

Type the following command at the OS/2 command prompt:

```
LINK @filename[.ext]
```

The @ symbol tells LINK that *filename* is a response file. If the file is not in the working directory, you must specify the path.

The file should contain responses in the same order as the LINK prompts. The file has the following order:

objectmodules
[*runfile*]
[*listfile*]

[*libraries*]
[*definitionsfile*]

Each response to a prompt must begin on a new line. If you choose not to specify one line of the response file, LINK will create the default for that response. For example, if you omit the *listfile* line, the default response supplied by LINK is NUL.MAP.

Responses that are longer than one line can be entered by typing a plus (+) sign at the end of each incomplete line, but the complete response must be less than 128 characters.

A semicolon can be used on any response line to instruct LINK to supply the default response for all remaining prompts. LINK then ignores the remainder of the responses in the file.

End the response file with either a semicolon or a carriage return/line feed combination. Otherwise, LINK displays the last line of the file and waits for you to press the Enter key.

When specifying more than one option, group them at the end of a response line or distribute them among several response lines. Every option must begin with a slash (/).

You can specify LINK options on the same line as your response or on separate lines.

If the file does not contain responses for all the prompts, LINK displays the appropriate prompts and waits for you to enter responses.

Example of a Response File

The response file in the following example tells LINK to generate an executable file called MODA.EXE from the four object modules MODA, MODB, MODC, and STARTUP.

Specifying the file name abc tells LINK to generate a list (map) file named abc.MAP. Adding the /MAP option tells LINK to include the application's public symbols in the map file.

Specifying the library search path tells LINK to search the subdirectory called lib on the b: drive for the library file called math.

```
moda+modb+modc+startup /MAP
moda.exe
abc
b:\lib\math
;
```

The Temporary Disk File

LINK creates output files during the linking process. These temporary files use physical memory first, but are written to a file on disk if insufficient memory is available. The disk files are created in the working directory, and the user receives the following message:

```
Temporary file name has been created
Do not change diskette in drive letter:
```

LINK supplies a name for the temporary file. The file is deleted automatically when the executable file has been created. The *Do not change...* message does not appear if a fixed disk drive is used for the LINK process.

The operation of LINK is unpredictable if you remove the diskette containing the temporary disk file during linking. If LINK ends unexpectedly, you may see the error message:

```
LINK: fatal error L1087:
unexpected end-of-file on scratch file
```

If this occurs, you must restart LINK.

LINK Output as a Debugging Aid

Map File for the OS/2 Environment

Following is an example of a map file created for the OS/2 environment:

```
PROGRAM_A

Start     Length     Name            Class

0001:0000 02C24H     _TEXT           CODE
0001:2C30 02502H     EMULATOR_TEXT   CODE
0001:5132 00000H     C_ETEXT         ENDCODE
0002:0000 00170H     EMULATOR_DATA   FAR_DATA
0003:0000 00036H     NULL            BEGDATA
0003:0036 00708H     _DATA           DATA
```

The *Start* column contains the address of the first byte in each segment, in the form *segment number:offset*. The segment numbers are indexes to the segment table of the executable file, and start at 1.

The *Length* column gives the length of the segment in bytes.

The *Name* column gives the name of each segment.

The *Class* column gives the class name of each segment.

Group information has the following form:

```
Origin        Group
0003:0        DGROUP
```

The *Origin* column contains the starting addresses ofthe groups of segments.

The *Group* column gives the names of the groups of segments.

The program entry point address is given at the end of the map file.

```
Program entry point at 0001:02A0
```

Map File for the DOS Environment

Following is an example of a map file created for the DOS environment:

```
Start  Stop   Length   Name            Class

00000H 02B86H 02B87H   _TEXT           CODE
02B90H 05091H 02502H   EMULATOR_TEXT   CODE
05092H 05092H 00000H   C_ETEXT         ENDCODE
050A0H 0520FH 00170H   EMULATOR_DATA   FAR_DATA
05210H 05245H 00036H   NULL            BEGDATA
05246H 05761H 0051CH   _DATA           DATA
```

The *Start* column contains the address of the first byte in each segment. The number shown is the offset from the beginning of the program.

The *Stop* column gives the address of the last byte in each segment.

The *Length* column gives the length of each segment in bytes.

The *Name* column gives the name of each segment.

The *Class* column gives the class name of each segment.

Group information has the following form:

```
Origin        Group
0521:0        DGROUP
```

The *Origin* column contains the starting addresses of the groups of segments.

The *Group* column contains the names of the groups of segments.

The program entry point address is given at the end of the map file.

```
Program entry point at 0000:02A0
```

Examples of Public Symbol Listings

```
Address             Publics by Name

0003:071A           $i8_implicit_exp
0003:0718           $i8_inpbas
0001:287C           $i8_input
0003:0719           $i8_input_ws
0001:0CF6           $i8_output

Address             Publics by Value

0000:0000    Imp    DOSWRITE      (DOSCALLS.138)
0000:0000    Imp    DOSDEVCONFIG  (DOSCALLS.52)
0000:0000    Imp    DOSEXIT       (DOSCALLS.5)
0001:0010           _main
0001:01B0           _countwords
0001:0238           _analyze
0001:02A0           _astart
0001:0368           _cintDIV
```

The form is the same for programs that execute in either the OS/2 or DOS environments. However, the number to the left of the colon in the symbol address has a different meaning in each environment. In the OS/2 environment, it is a segment number. In the DOS environment, it is the offset from the beginning of the program.

The number to the right of the colon is the segment offset.

The first three symbols (described in the first three lines) in the second example are imported public symbols and appear in map files created for programs that run only in the OS/2 environment.

Finding Errors

For effective use of LINK, see your language manual for information about defining application segments.

The information in Table 6-1 can help you correct errors encountered during the linking process.

Item	Limit
Symbol table	256KB
Load-time relocations (for OS/2 applications)	Default is 32KB. If /EXEPACK is used, the maximum is 512KB.
Public symbols	The range 7700—8700 can be used as a guideline for the maximum number of public symbols allowed; the actual maximum depends on the application.
External symbols per module	1023
Groups	Maximum number is 21, but LINK always defines DGROUP so the actual maximum is 20.
Segments	128 by default; however, the number can be set as high as 3072 by using LINK's /SEGMENTS option.
Libraries	32
Group definitions per module	21
Segments per module	255
Stack	64KB

Table 6-1. LINK Limits

Intersegment Calls

In many systems, such as DOS, intersegment references must be resolved within the application's executable file. This means that LINK resolves all references and sends a single executable file containing all called subroutines to the loader. The loader brings the executable file into memory and fixes it up to ensure the intersegment references are valid. In the OS/2 environment, resolution of external references can be delayed until load time or run time.

In the OS/2 environment, an application can reference segments that are not part of its executable file. The LINK utility reserves place holders in the executable file for information about external segments. Until load time, the code, data, and stack segments that comprise the application's executable file reside on disk. When the loader receives a request to execute the application, it must build a local descriptor table (LDT) and allocate memory for the application to run. It also fills in the addresses and other information about the application's external segments.

Chapter 7. LINK Options

When including LINK options, follow these guidelines.

- You can specify an option on any line before the last comma at the end of the LINK command or response line.
- Every option must begin with the slash (/) character, even if other options appear before it on the line.
- You can abbreviate option names as long as the abbreviations contain enough letters to distinguish the specified option from other options. Minimum abbreviations are listed with the description for each option.
- LINK does not recognize spaces between characters, nor transposed letters.

Entry of Numeric Parameters

Numeric parameters in LINK options can be entered in decimal, hexadecimal, or octal. The format follows the C language conventions, which are:

Decimal	Any number that begins with anything other than a 0 digit. For example: 1, 65536, 2084, 234.
Hexadecimal	Any number that begins with 0x and contains the digits 0 through 9, and the letters A through F. For example: 0xFFF, 0x10.
Octal	Any number that begins with the digit 0 and contains only the digits 0 through 7. For example: 010, 05000, 0777.

Summary of LINK Options

The LINK options are:

Option	*Description*
/ALIGNMENT	Sets segment alignment factor.
/CODEVIEW	Includes debugging information for the CodeView debugger.
/CPARMAXALLOC	Changes value of maximum number of reserved paragraphs.
/DOSSEG	Forces ordering of segments.
/DSALLOCATE	Controls data loading.
/EXEPACK	Packs executable files.
/FARCALLTRANSLATION	Optimizes intersegment far calls.
/HELP	Writes a list of the available options to the screen.
/HIGH	Controls loading the run file.
/INFORMATION	Displays information about the linking process.
/LINENUMBERS	Copies line numbers to the map file.
/MAP	Lists all public symbols in your program.
/NODEFAULTLIBRARYSEARCH	Ignores default libraries.
/NOEXTENDEDDICTIONARYSEARCH	Prevents LINK from searching the extended dictionary.

/NOFARCALLTRANSLATION	Disables far call translations.
/NOGROUPASSOCIATION	Provides compatibility with previous compiler versions.
/NOIGNORECASE	Differentiates between uppercase and lowercase letters.
/NOPACKCODE	Disables code segment packing.
/OVERLAYINTERRUPT	Sets the overlay interrupt number.
/PACKCODE	Packs code segments.
/PACKDATA	Packs data segments.
/PAUSE	Pauses to change diskettes.
/SEGMENTS	Sets the maximum number of segments.
/STACK	Sets the stack size.
/WARNFIXUP	Warns of incorrect offset.

Option Descriptions

A description of the individual options follows.

/ALIGNMENT
Aligning Segments

Purpose

To set the segment alignment factor in the executable file to a specified number of bytes.

Format

/ALIGNMENT:*number*

The minimum abbreviation is **/A**. *number* can be a decimal, hexadecimal, or octal number.

Defaults

The default alignment factor is 512.

Restrictions

- *number* must be a power of 2.
- This option is valid only for code that is linked to run in the OS/2 environment.

Remarks

LINK aligns segments to ensure proper loading. Aligning a segment means moving its address to the next address boundary divisible by the alignment factor.

We recommend an alignment factor of 16 for Presentation Manager applications.

/CODEVIEW
Preparing Files for CodeView

Purpose

To include symbolic debugging information for CodeView™ in the executable file.[1]

Format

/CODEVIEW

The minimum abbreviation is **/CO**.

Restrictions

/CODEVIEW cannot be used with /EXEPACK.

/CPARMAXALLOC
Reserving Paragraph Space

Purpose

To change the default value of the MAXALLOC field.

Format

/CPARMAXALLOC:*number*

The minimum abbreviation is **/CP**.

Defaults

The default for the MAXALLOC field is 65535 (decimal), that is, 64K minus 1.

Restrictions

- This option is valid only for code linked to run in the DOS environment or in the DOS mode of OS/2.
- The maximum number of paragraphs reserved for an application is determined by the value of the MAXALLOC field at offset 0x0C in the .EXE header.
- If the value you specify is less than the computed value of MINALLOC (at offset 0x0A), LINK uses the value of MINALLOC instead.

[1] Trademark of Microsoft Corporation

Remarks

A paragraph is the smallest storage unit (16 bytes) addressable by a segment register. This field controls the maximum number of paragraphs reserved in memory for your application.

You can reset the default to any number between 1 and 65535 (decimal) using decimal, hexadecimal, or octal notation. Changing the default is helpful because reserving all available memory does not increase the performance of your application.

If your application invokes another application, you need to reserve memory for that application.

/DOSSEG
Ordering Segments

Purpose

To order segments in the following sequence:

- All segments having a class name ending in CODE
- All other segments outside of DGROUP
- DGROUP segments in the following order:
  - Any segments of class BEGDATA reserved.
  - Any segments not of class BEGDATA, BSS, or STACK
  - Segments of class BSS
  - Segments of class STACK

Format

/DOSSEG

The minimum abbreviation is **/DO**.

Remarks

Linking with the standard C/2 run-time libraries automatically enables the /DOSSEG option by comment record in the startup module.

Using this option inserts 16 bytes of NULLs in front of the segment named _TEXT, which if present is a requirement for C/2 runtime support.

/DSALLOCATE
Controlling Data Loading

Purpose

To load all data starting at the high end of the data segment.

Format

/DSALLOCATE

The minimum abbreviation is **/DS**.

Defaults

By default, LINK loads all data starting at the low end of the data segment. At run time, LINK sets the data segment pointer to the lowest possible address to allow the entire data segment to be used.

Restrictions

This option is valid only for code linked to run in the DOS environment or in the DOS mode of OS/2.

Remarks

Set the data segment pointer at run time to the lowest data segment address containing program data.

The /DSALLOCATE option typically is used with the /HIGH option to take advantage of unused memory within a data segment. You can reserve available memory below the area specifically reserved for DGROUP by using the same data segment pointer.

/EXEPACK
Packing Executable Files

Purpose

To remove sequences of repeated bytes (typically nulls) and to make most efficient use of the load-time relocation table before creating a DOS executable file.

Format

/EXEPACK

The minimum abbreviation is **/E**.

Restrictions

- This option is valid only for code linked to run in the DOS environment or in the DOS mode of OS/2.
- /EXEPACK cannot be used with /CODEVIEW because symbolic debuggers cannot be used with packed files.

Remarks

OS/2 executable files are always compressed (packed). Executable files linked with this option are usually smaller and load faster than files linked without using this option.

The /EXEPACK option does not always save disk space and may sometimes increase file size. Applications having a large number of load-time relocations (500 or more) or long streams of repeated characters are usually shorter if packed.

Note: The LINK options /EXEPACK and /HIGH are mutually exclusive.

/FARCALLTRANSLATION
Optimizing Translation of Far Calls

Purpose

To optimize translation of intersegment far calls using the sequence:

```
NOP
PUSH CS
CALL NEAR address
```

Format

/FARCALLTRANSLATION

The minimum abbreviation is **/FAR**.

Defaults

The default option is /NOFARCALLTRANSLATION.

Restrictions

This option is valid only for code linked to run in the OS/2 environment.

In translating far calls, LINK may mistake a byte that uses the value 0x9A as a constant for a far call.

Remarks

Using this option for most medium and large model applications makes the executable file smaller, thus saving load-time.

/HELP
Viewing the Options List

Purpose

To write a list of the available options to the screen.

Format

/HELP

The minimum abbreviation is **/HE**.

Remarks

Do not give a file name when using the /HELP option.

/HIGH
Controlling Where the Executable File is Loaded

Purpose

To load the executable file as high as possible in memory without overlaying the transient portion of COMMAND.COM.

Format

/HIGH

The minimum abbreviation is **/HI**.

Defaults

COMMAND.COM is loaded at the highest area of memory; the executable file, at the lowest end.

Restrictions

- This option is valid only for code linked to run in the DOS environment or in the DOS mode of OS/2.
- The /HIGH option should not be used with programs written in C.

Remarks

Use the /HIGH option in association with the /DSALLOCATE option.

Note: The LINK options /EXEPACK and /HIGH are mutually exclusive.

/INFORMATION
Displaying Information About the Linking Process

Purpose

To display information, such as the names of files being linked during the linking process.

Format

/INFORMATION

The minimum abbreviation is **/I**.

Remarks

/INFORMATION is useful for debugging.

/LINENUMBERS
Copying Line Numbers to the Map File

Purpose

To copy the line number of the starting address of each instruction in the application source code.

Format

/LINENUMBERS

The minimum abbreviation is **/LI**.

Restrictions

The numbering of lines of source code is supported by most high-level languages, but not Macro Assembler. To obtain the line numbers associated with the starting address of each instruction, you must request that LINK generate a map file.

Remarks

If an object file has no line number information, LINK ignores the /LINENUMBERS option.

If you do not specify a map file in a LINK command, you can still use the /LINENUMBERS option to create a map file. Place the option at or before the List File [NUL.MAP]: prompt. LINK gives the map file the same file name as the first object file specified in the command and gives it the default extension .MAP.

/MAP
Producing a Public Symbol Map

Purpose

To produce a listing of all the public symbols declared in your application. This list is copied to the map file created by LINK.

Format

/MAP[*:number*]

The minimum abbreviation is **/M**. *number* can be a decimal, hexadecimal, or octal number.

Defaults

If you do not specify a value for *number*, the default is 2048.

Restrictions

1 through 32767 are valid values.

Remarks

The *number* parameter specifies the maximum number of public symbols that LINK can sort in the map file.

Specifying a number generates a map file with the public symbols sorted by address rather than by name. If you do not need the list of public symbols sorted by name, you can tell LINK not to include it. When you specify the /MAP option, include a number large enough to accommodate only the number of public symbols in the application.

If you do not request a map file in a LINK command, you can use the /MAP option to create a map file. LINK gives the map file the same name as the first object file specified in the command and the default file extension .MAP.

For a description and examples of the format of a map file, see "LINK Output as a Debugging Aid" on page 6-5.

/NODEFAULTLIBRARYSEARCH Ignoring Default Libraries

Purpose

To tell LINK to ignore any default libraries named in object files.

Format

/NODEFAULTLIBRARYSEARCH

The minimum abbreviation is **/NOD**.

Remarks

High-level language compilers sometimes include library names in the object files they generate. Using this option ensures that only the libraries you name are linked with your object files.

/NOEXTENDEDDICTIONARYSEARCH Preventing Extended Dictionary Search

Purpose

To prevent LINK from searching the extended dictionary. The extended dictionary is a list of inter-library dependencies. For example, suppose *FOO*, *X*, and *Y* are procedure names; after LINK finds *FOO* it knows from the extended dictionary that *X* and *Y* are required from the same library.

Format

/NOEXTENDEDDICTIONARYSEARCH

The minimum abbreviation is **/NOE**.

Defaults

The extended dictionary is searched by default.

Restrictions

None.

Remarks

The extended dictionary speeds up library searches, so in general /NOE causes linking to take longer. A library without an extended dictionary is still a valid library.

/NOE is also used to redefine a symbol that is already defined in a library. In this case, if you fail to use the /NOEXTENDEDDICTIONARYSEARCH switch, a *symbol multiply defined* error occurs.

/NOFARCALLTRANSLATION
Disabling Far Call Translations

Purpose

To disable translation of intersegment far calls.

Format

/NOFARCALLTRANSLATION

The minimum abbreviation is **/NOF**.

Defaults

This is the default (see /FARCALLTRANSLATION).

Restrictions

This option is valid only for the OS/2 environment.

/NOGROUPASSOCIATION
Maintaining Compatibility

Purpose

To process a specific set of fix-up routines for compatibility with earlier versions of LINK and language compilers.

Format

/NOGROUPASSOCIATION

The minimum abbreviation is **/NOG**.

Restrictions

- This option is valid only for code linked to run in the DOS environment or in the DOS mode of OS/2.
- It should not be used with applications written in C.

Remarks

None.

/NOIGNORECASE
Recognizing Uppercase and Lowercase

Purpose

To recognize upper-case and lower-case letters as distinct characters when used in symbol names.

Format

/NOIGNORECASE

The minimum abbreviation is **/NOI**.

Examples

LINK recognizes uppercase and lowercase letters as identical. For example, it considers *TWO*, *Two*, and *two* the same. Using this option tells LINK to recognize the three symbol names as distinct.

Remarks

This option is typically used with object files created by high-level language compilers. Some compilers recognize upper-case and lower-case letters as distinct letters and assume that LINK does the same.

/NOPACKCODE
Disabling Code Segment Packing

Purpose

To tell LINK not to pack neighboring logical code segments into one physical segment.

Format

/NOPACKCODE

The minimum abbreviation is **/NOP**.

Defaults

The default option is /PACKCODE.

Restrictions

This option is valid only for code linked to run in the OS/2 environment.

Remarks

For more information on packing, see the /PACKCODE option.

/OVERLAYINTERRUPT
Overriding the DOS Interrupt Number

Purpose

To override the default DOS interrupt number for passing control to overlays.

Format

/OVERLAYINTERRUPT:*number*

The minimum abbreviation is **/O**. *number* can be a decimal number from 0 to 255, an octal number from 0 to 0377, or a hexadecimal number from 0 to 0xFF.

Defaults

0x3F is the default DOS interrupt number.

Restrictions

This option is valid only for code linked to run in the DOS environment or in the DOS mode of OS/2.

Remarks

Numbers that conflict with DOS interrupts are not prohibited, but we do not recommend using them.

/PACKCODE
Packing Code Segments

Purpose

To tell LINK to pack neighboring logical code segments into one physical segment.

Format

/PACKCODE[*:packlimit*]

The minimum abbreviation is **/PACKC**. *packlimit* is the maximum size of the segment, in bytes.

Defaults

- The option /PACKCODE is the default. /NOPACKCODE should be used to override /PACKCODE.
- If /PACKCODE is entered without a *packlimit*, LINK uses the default of 65536.

Restrictions

This option is valid only for code linked to run in the OS/2 environment.

Remarks

The optional *packlimit* is the limit at which to stop packing.

Note: Code segments 65501 – 65536 in length may be unreliable on the 80286 microprocessor. Use of these segments will generate the following warning message:

```
LINK: warning L4011:
PACKCODE value exceeding 65500 unreliable.
```

/PACKDATA
Packing Data Segments

Purpose

Tells LINK to pack neighboring logical data segments into one physical segment.

Format

/PACKDATA[*:packlimit*]

The minimum abbreviation is **/PACKD**.

Defaults

By default, LINK does not try to pack neighboring logical data segments into one physical segment. The *packlimit* is the limit at which to stop packing. If no number is given, LINK uses 65536.

Restrictions

This option is valid for OS/2 executable programs only.

Remarks

Consider using this option if you have a large-model application containing many files and you receive the following message: L1073-file-segment limit exceeded.

/PAUSE
Pausing to Change Disks

Purpose

To tell LINK to pause before writing the executable file to a disk file, you can change diskettes.

Format

/PAUSE

The minimum abbreviation is **/PAU**.

Remarks

If you use the /PAUSE option, LINK displays the following message before creating the executable file:

About to generate .EXE file
Change diskette in drive *letter* and press Enter

where *letter* is the correct drive name. This message appears after LINK has read data from the object and library files and after it has written data to a map file. LINK resumes processing when you press Enter.

After LINK writes the executable file to a disk file, the following message appears:

Please replace original diskette in drive *letter*
and press Enter

Note: Do not remove the diskette containing the temporary file. If the following message appears, press Ctrl and Break keys to end the LINK session.

```
LINK: fatal error L1087:
unexpected end-of-file on scratch file
```

Rearrange your files so LINK can write the temporary file and the executable file to the same diskette and try again.

/SEGMENTS
Setting the Maximum Number of Segments

Purpose
To process no more than a specified number of segments for each application.

Format

/SEGMENTS:*number*

The minimum abbreviation is **/SE**. *number* can be any integer value in the range 1 to 3072.

Defaults

- The /SEGMENTS option bypasses the default limit of 128 segments.
- If you do not specify /SEGMENTS, LINK reserves enough memory to process up to 128 segments. If your program has more than 128 segments, you must set the segment limit higher to increase the number of segments that LINK can process.

Examples

This example sets the segment limit to 192:

```
LINK file/SE:192,,;
```

This example sets the segment limit to 255 (0xFF):

```
LINK moda+modb,run/SEGMENTS:0xff,ab,em+mlibfp;
```

Remarks

Set the segment limit lower if you get the following LINK error message:

```
LINK: error L1054:
requested segment limit too high.
```

/STACK
Setting the Stack Size

Purpose
To set the application stack to a specified number of bytes.

Format

/STACK:*size*

The minimum abbreviation is **/ST**. *size* can be any positive integer value in the range 0 to 65534.

Examples

This example sets the stack size to 512 bytes:

```
LINK file/STACK:512,,;
```

This example sets the stack size to 255 (0xFF) bytes:

```
LINK moda+modb,run/ST:0xFF,ab,\lib\start;
```

This example sets the stack size to 24 (030 octal) bytes:

```
LINK startup+file/ST:030,,;
```

Remarks

LINK automatically calculates the size of the stack allocated to an application. The stack size is based on the size of the stack segments specified in the object files. If you specify the /STACK option, LINK uses the specified *size* in place of the value it calculated.

The stack size can also be changed using STACKSIZE statement in the module definition file.

/WARNFIXUP
Warning of Incorrect Offset

Purpose

To issue a warning for each segment relative fix-up of location-type offset. This option should be used specifically if the segment is contained within a group, but is not at the beginning of the group.

Format

/WARNFIXUP

The minimum abbreviation is **/W**.

Remarks

LINK includes the displacement of the segment from the group in determining the final value of the fix-up. This is the opposite of what happens with DOS executable files.

Note valid for OS/2 executable files only.

Chapter 8. Module Definition File Statements

The OS/2 Linker (LINK) accepts a text file called a module definition (.DEF) that is written with any ASCII text editor. The module definition file is optional for full-screen OS/2, full-screen DOS, and family applications. The statements and keywords in the module definition file provide LINK with information about the executable file or dynamic link library.

The module definition file statements define the attributes of code and data segments, imported functions, and heap and stack sizes. Many of these statements have appropriate defaults, which are included in the following section, so you need not declare them. Some of the options are appropriate only for specific execution environments, and they are identified.

Summary of Statements

Statement	Description
CODE	Defines default attributes for code segments.
DATA	Defines default attributes for data segments.
DESCRIPTION	Inserts text into a program module.
EXPORTS	Defines exported functions from dynamic link libraries.
HEAPSIZE	Defines heap size in bytes.
IMPORTS	Defines imported functions from dynamic link libraries.
LIBRARY	Declares a dynamic link library.
NAME	Declares a program module.
NEWFILES	Designates the program supports non 8.3 file names.
OLD	Directs preservation of earlier ordinals.
PROTMODE	Declares program to run in the OS/2 environment only.
SEGMENTS	Defines attributes for code and data segments on a per segment basis.
STACKSIZE	Defines stack size in bytes.
STUB	Appends DOS executable file to the OS/2 program module.

Notes:

- LINK ignores all comment lines preceded by semicolons in the module definition file.
- The keywords in the statements in the module definition file must be in uppercase.

Statement Descriptions

A complete list of the module definition file statements follows.

CODE
Defining the Code Segment Default Attributes

Purpose

This statement defines the default attributes of all code segments in the module.

Format

CODE [*load option*] [*execute option*] [*privilege option*][*conforming option*]

Remarks

The *load option* is an optional keyword specifying when the segment is to be loaded. It must be one of the following:

PRELOAD The segment is loaded immediately.

LOADONCALL The segment is loaded when called.

The default is LOADONCALL.

The *execute option* specifies whether the segment can be read as well as run. It must be one of the following:

EXECUTEONLY The segment can only be run.

EXECUTEREAD The segment can be run and read.

The default is EXECUTEREAD.

Note: If you use C to create a code segment of a dynamic link module and the source code contains a switch statement, do not specify EXECUTEONLY on the CODE statement of the dynamic link module definition file. Use the default execute option EXECUTEREAD for such segments. In many cases of the switch statement, the compiler generates a table lookup through the CODE segment. The table lookup method is taken for switch statements with more than six distinct cases when the ratio of number-of-cases to range-of-cases exceeds one-third.

The *privilege option* is an optional keyword specifying if the segment has I/O privilege. It must be one of the following:

IOPL The segments have I/O privilege.

NOIOPL The segments do not have I/O privilege.

The default is NOIOPL.

Conforming option is an optional keyword specifying the access rights to code segments. It must be one of the following:

CONFORMING Sets access rights to conforming.

NONCONFORMING Sets access rights to nonconforming.

The default is NONCONFORMING. Set the access rights to CONFORMING if the segment contains at least one routine that has all of the following qualities:

- Will be called from both IOPL and non-IOPL segments
- Will not be called through call gates
- Does not need I/O privilege to operate (that is, does not access ring 2 data, disable interrupts, or use the IN and OUT instructions).

DATA
Defining Data Segment Default Attributes

Purpose

This statement defines the default attributes of the data segments of the application. The automatic data segment contains the local stack and heap of the module.

Format

```
DATA [instance option] [shared option] [write option]
  [privilege option] [load option]
```

Remarks

The *instance option* is an optional keyword that describes the sharing of the automatic data segment, which is any group named DGROUP. It can be any one of the following:

NONE There is no automatic data segment.

SINGLE The automatic data segment is shared by all instances of the module (valid only for dynamic link libraries).

MULTIPLE The automatic data segment is copied for each instance of the module.

The default for *instance option* is MULTIPLE for program modules and SINGLE for dynamic link libraries.

The *shared option* is an optional keyword specifying the need to have a unique copy of the READWRITE data segments loaded for each process using the dynamic link library. Values are:

SHARED A single copy of each data segment is loaded.

NONSHARED A unique copy of each READWRITE data segment is loaded for each process using the dynamic link library.

The default is NONSHARED for program modules and SHARED for dynamic link libraries.

The *write option* is an optional keyword specifying whether the segment can be written to. It must be one of the following:

READONLY The segment can only be read from.

READWRITE The segment can be read from or written to.

The default is READWRITE.

The *privilege option* is an optional keyword specifying if the segment has I/O privilege. It must be one of the following:

IOPL The segments have I/O privilege.

NOIOPL The segments do not have I/O privilege.

The default is NOIOPL.

The *load option* is an optional keyword specifying when the segment is to be loaded. It must be one of the following:

PRELOAD The segment is loaded immediately.

LOADONCALL The segment is loaded when called.

If you give a CODE statement, the default is LOADONCALL.

DESCRIPTION
Inserting Text

Purpose

This statement inserts *text* into the module of the application. It is useful for embedding source control or copyright information.

Format

DESCRIPTION *text*

Remarks

The *text* is one or more ASCII characters. You must enclose the string in single quotation marks.

Examples

```
DESCRIPTION 'Template Application'
```

EXPORTS
Exporting Functions

Purpose

This statement defines the names and characteristics of the functions in the dynamic link library to export to other applications. The EXPORTS keyword marks the beginning of the definitions. Following the EXPORTS keyword are up to 3072 export definitions, each on a separate line.

Format

EXPORTS
exportname [*ordinal option*][RESIDENTNAME] [*iopl -parmwords*]

Remarks

The *exportname* is one or more ASCII characters defining the function name. It has the form:

entryname[=*internalname*]

The *entryname* is the name that other applications use to get access to the exported function. It is a required parameter.

The case of the *exportname* should match the case of the name as it exists in the .OBJ file. Failure to do so may prevent use of the /NOI switch on the linker.

The *internalname* defines the actual name of the function, if *entryname* is not the actual name. It is an optional parameter.

The *ordinal option* defines the ordinal value of the function. It has the form:

@ ordinal

where *ordinal* is an integer number specifying the ordinal value of the function. The ordinal value defines the index of the name of the function in the entry table of the dynamic link library. A blank space must precede the @ character. It is an optional parameter.

RESIDENTNAME indicates that the entry point name of the function is kept resident in memory. It is an optional parameter, applicable only when *ordinal* option is specified. DOS normally keeps name strings resident in memory, enabling it to rapidly resolve calls to frequently used entry points.

The *iopl-parmwords* is an optional numeric value that must be specified for functions that run with I/O privilege. Each such function is allocated a 512-byte stack. When the function is invoked, the number of parameters (words) specified by *iopl-parmwords* are copied from the caller's stack to the new stack.

Examples

```
EXPORTS
 SampleRead= @1 8
 StringIn=strl @2
 CharTest @3
```

HEAPSIZE
Defining Local Storage

Purpose

This statement defines the number of bytes needed by the application for its local heap. An application uses the local heap whenever it reserves local storage. The default heapsize is 0.

Format

HEAPSIZE *bytes*

Remarks

The *bytes* parameter is an integer number specifying the heap size in bytes. It must not exceed 65536 (64KB), the size of a single physical segment. Note that we define *heap* as a collection of memory blocks.

Examples

```
HEAPSIZE 4096
```

IMPORTS
Importing Functions

Purpose

This statement defines the names and attributes of functions to be imported from existing dynamic link modules. The IMPORTS keyword marks the beginning of the definitions. There can be any number of IMPORTS statements, each on a separate line.

Format

```
IMPORTS
[internal-name=] libraryname.entry
```

Remarks

The *internal-name* is an optional specification of one or more ASCII characters. It specifies the name that the application uses to call the function. It must be a unique identifier. The *libraryname* is the name of the dynamic link library containing the function. The *entry* specifies the function to import. It is one of the following:

entryname The actual name of the function.

entryordinal The ordinal value of the function; corresponds to the entry point in the dynamic link library. For each entry of this type you must specify an internal-name.

Note: *.entryname* references to DOSCALLS are not supported and will fail during the module load. Therefore, rather than directly importing OS/2 functions, link with OS2.LIB.

If *internal-name* is not given, the *entry* must be *entryname*, which is used for the internal name.

Examples

```
IMPORTS
   Sample.SampleRead
   write2hex=Sample.SampleWrite
   read=Read.1
```

LIBRARY
Naming Library Modules

Purpose

This statement specifies the creation of a dynamic link library. It also specifies the type of library initialization required for the dynamic link library.

Note: The LIBRARY statement tells LINK to build a dynamic link library. Do not associate the LIBRARY statement with .LIB files that are libraries of object modules.

Format

LIBRARY[*libraryname*] [*initialization-type*]

Remarks

The *libraryname* defines the name of the dynamic link library. Although the *libraryname* is not normally specified, it can be up to eight characters. The actual library name created by the linker is based on the external name of the executable file. The linker uses the *libraryname* to validate the library name being generated. If the internal library name does not match the one specified by *libraryname*, the linker issues a warning message. When used, LIBRARY must be the first statement specified in the module definition file. Once a dynamic link library has been loaded, OS/2 knows it by the internal library name.

Initialization-type is an optional keyword that specifies the type of initialization required by the dynamic link library. If no library initialization routine is defined for the library, this keyword is ignored. It must be one of the following:

INITGLOBAL The library module initialization routine is called once when the library module is initially loaded.

INITINSTANCE The library module initialization routine is called once for each process that gains access to the library module.

The default is INITGLOBAL.

Note: The starting address of the module is determined by the object files and refers to a function that must conform to the library initialization convention that DOS requires.

Comments

The LIBRARY statement is mutually exclusive with the NAME statement. If the LIBRARY statement is used, it must appear as the first statement in the definitions file. If neither LIBRARY nor NAME is included in a module definition file, the default is NAME.

Examples

```
LIBRARY User
```

NAME
Naming Executable Modules

Purpose

This statement specifies creation of a program module.

Format

NAME[*modulename*] [*applicationtype*]

Remarks

The *modulename* is optional. The actual module name created by the linker is based on the external name of the executable file. The linker uses the *modulename* to validate the module name being generated. If the internal module name does not match *modulename*, the linker issues a warning message. *Applicationtype* is optional and defines the type of application being linked. This information is kept in the executable header and is used by the Presentation Manager during program load and execution. If specified, it must be one of the following:

WINDOWAPI Uses the API provided by Presentation Manager. It must be executed in a Presentation Manager window.

WINDOWCOMPAT Runs (compatible) in a Presentation Manager window or in a separate screen group. An application is of this type if it uses the proper subset of OS/2 VIO, KBD, and MOU functions.

NOTWINDOWCOMPAT Does not run (not compatible) in a Presentation Manager window. It must execute in a separate screen group. All DOS mode applications are of this type. Those that use the VIO, KBD, and MOU functions not compatible with Presentation Manager also are of this type.

If *applicationtype* is not provided, the executable header will indicate that the application type was unspecified.

Examples

```
NAME Calendar WINDOWCOMPAT
```

Comments

The NAME statement is mutually exclusive with the LIBRARY statement. If the NAME statement is used, it must appear as the first statement in the definitions file. If neither NAME nor LIBRARY is included in a module definition file, the default is NAME.

NEWFILES
Supporting Non 8.3 File Names

Purpose
This statement directs LINK to set the newfiles bit in the PC/DOS executable file header. It indicates that the module supports non 8.3 file names. This bit is meaningless in real mode, on DOS, and on previous versions of OS/2.

Format
```
NEWFILES
```

Remarks

NEWFILES designates that the module supports non 8.3 file names. 8.3 DOS file names have no more than 8 characters in the name component and no more than 3 characters in the extension. Non 8.3 file names may be up to 255 bytes long, may have multiple dots, and may contain characters which are illegal in 8.3 DOS file names.

Programs written for OS/2 Version 1.2 Installable File Systems should set this bit. Bound programs that have this bit set will see non 8.3 file names in protect mode and 8.3 only file names in real mode.

This bit has meaning when attached to program modules, not when attached to dynamic link library (.DLL) files . Whether the program sees non 8.3 file names is entirely dependent on the value of its NEWFILES bit and the bit's effect extends into calls to any dynamic link library (.DLL) files. For a complete discussion on new long file names, see the *Programming Guide*.

Note: Do not use NEWFILES for the following:

- PC/DOS or 3xbox programs
- Programs which do not support NEWFILES naming rules.

Examples
```
NEWFILES
```

OLD
Preserving Export Ordinals

Purpose
This statement preserves export ordinals across successive versions of a dynamic link library.

Format
```
OLD 'libraryname'
```

Remarks

The *libraryname* is the name of the dynamic link library to be used for extracting export ordinals. The *libraryname* must be within single quotation marks.

Exported names in this library that match exported names in the OLD library are assigned ordinal values from the OLD library, unless:

1. The name in the OLD library did not have an assigned ordinal, or
2. An ordinal was explicitly assigned to a name in this library.

Note: If the linker cannot find *libraryname* in the current directory, it looks in the directories listed in the LIB PATH environment variable.

PROTMODE
Setting OS/2 Environment

Purpose

This statement causes the linker to set the OS/2-mode-only bit in the file header of the executable (.EXE) file. Without a PROTMODE statement, the OS/2-mode-only bit is not set.

By default, LINK does not set the OS/2-mode-only bit.

Format

```
PROTMODE
```

Remarks

If the linker recognizes floating-point instructions in the object module and the OS/2-mode-only bit is not requested to be set, the linker produces run-time relocations of type OSFIXUP. If you use the resultant executable program as input to the BIND utility, BIND can emulate the floating point instructions for the DOS mode if no coprocessor is present. If you request the OS/2-mode-only bit be set, the linker does not produce OSFIXUP relocations and the BIND utility cannot be used to emulate the floating point instructions for DOS mode.

For programs that extensively use floating point instructions, the amount of space in the executable file that OSFIXUP relocations occupy can be significant. By using a definitions file containing the PROTMODE statement, you can save space in the executable file. (Do this only if you intend to run the program only in OS/2 mode.)

SEGMENTS
Defining Segments

Purpose

This statement defines code and data segment attributes on a per-segment basis. The parameters specified override the defaults on the CODE and DATA statements. The SEGMENTS statement marks the beginning of the definitions. You can give any number of segment definitions, each on a separate line.

Format

```
SEGMENTS['']segmentname['] [class option] [segflags]
```

Remarks

Each segment definition consists of a combination of the parameters which follow.

The *segmentname* is a character string naming the new segment. Optionally, you can enclose the *segmentname* in single quotation marks. If the *segmentname* is CODE or DATA or any other definitions file keyword, you must enclose it in single quotations marks to avoid conflict with the CODE and DATA keywords.

The *class option* is an optional keyword specifying the class of the segment:

CLASS '*classname*'

Note: If you do not give a class name, the linker assumes class CODE. Any segment whose class name ends in CODE (case insensitive) is given the type CODE by the linker. If a segment is defined here without a CLASS directive, it is recognized as a code segment.

The *segflags* is any combination of the following options that are described under the CODE and DATA keywords above:

Option	Default
SHARED, NONSHARED	NONSHARED
PRELOAD, LOADONCALL	LOADONCALL
EXECUTEONLY, EXECUTEREAD (for code segments only)	EXECUTEREAD
READONLY, READWRITE (for data segments)	READWRITE
IOPL, NOIOPL	NOIOPL
CONFORMING, NONCONFORMING (for code segments only)	NONCONFORMING

Examples

```
SEGMENTS
     CSEG  LOADONCALL
     CSEG2 EXECUTEREAD
     DSEG1 READONLY IOPL
     DSEG2 SHARED
```

STACKSIZE
Defining Local Stack

Purpose

This statement defines the number of bytes needed by the application for its local stack. An application uses the local stack whenever it calls its own functions. A minimum stack size of 4096 bytes is recommended. The default stack size is zero if the application makes no function calls. Otherwise, it is 4096. The maximum stack size permitted is 65535 bytes.

Format

STACKSIZE *bytes*

Remarks

The *bytes* parameter is an integer specifying the stack size in bytes.

Examples

```
STACKSIZE 4096
```

STUB
Adding an Executable File to a Module

Purpose

This statement adds the DOS mode executable file named in *filename* to the beginning of the OS/2 mode module being created.

Format

```
STUB 'filename'
```

Remarks

The filename is the name of the DOS mode executable file to add to the module. The name must have the DOS filename format and be enclosed in single quotation marks. The stub is started if the OS/2 mode program is run in the DOS mode. The OS2STUB.EXE file is provided in the tools package and can be used as the filename.

Appendix A. Error Messages

Introduction

This appendix lists the error messages that may be encountered when building an application. Included is a brief description of the action required to correct the error. Error messages for the following utilities are listed:

- LINK
- RC (Resource Compiler)
- BIND
- IMPLIB
- MSGBIND

LINK Error Messages

Format of Error Messages

There are three types of LINK error messages:

- *Fatal errors* cause LINK to stop running. They have the following format:

  location: **error L1***xxx*:
  message text

- *Nonfatal errors* indicate problems in the executable file. LINK produces the executable file and sets the error bit in the header for the OS/2 environment. This means that the executable file cannot be run from OS/2. Nonfatal error messages have the following format:

  location: **error L2***xxx*:
  message text

- *Warnings* indicate possible problems in the executable file. LINK produces the executable file, but does not set the error-bit in the header for the OS/2 environment. Warnings have the following format:

  location: **error L4***xxx*:
  message text

In all these messages, *location* is the input file associated with the error, or it is LINK itself if there is no input file. The *message text* is the actual text message that LINK generates. When the input file is a module definition file, the line number is included, as in this example:

```
myfile.def(3): fatal error L1030:
missing internal name
```

When the input file is an object file or library file and has a module name, the module name is enclosed in parentheses, as in the following examples:

```
SLIBCE.LIB(_file)
MAIN.OBJ(main.c)
TEXT.OBJ
```

Error Message Descriptions

L1001 *option*: **option name ambiguous**

Explanation: A unique option name does not appear after the option indicator (/).

Example: The command

```
LINK /N main;
```

produces this error because LINK cannot tell which of the five options beginning with the letter N is intended.

Action: Retry using the correct minimum option abbreviation.

L1002 *option*: **unrecognized option name**

Explanation: An unrecognized character follows the option indicator (/).

Example: The command

```
LINK /ABCDEF main;
```

produces this error because /ABCDEF is not a valid option.

Action: Retry using a valid option.

L1003 *option*: **MAP symbol limit too high**

Explanation: The specified symbol limit value following the /MAP option is greater than 32767, or there is not enough memory to increase the limit to the requested value.

Action: Retry with a lower symbol limit.

L1004 *option*: **invalid numeric value**

Explanation: An incorrect value appeared for one of the LINK options. This may be because a character string has been entered for an option that requires a numeric value.

Action: Retry with a numeric value.

L1005 *option*: **packing limit exceeds 65536 bytes**

Explanation: The number following the /PACKCODE option is greater than 65536.

Action: Retry with a valid number.

L1006 *option*: **stack size exceeds 65534 bytes**

Explanation: The size you specified for the stack in the /STACK option of the link command is more than 65534 bytes.

Action: Retry with a stack size of less than 65534 bytes.

L1007 *option*: **interrupt number exceeds 255**

Explanation: You gave a number greater than 255 as a value for the /OVERLAYINTERRUPT option.

Action: Retry with a value in the range 0 to 255.

L1008 *option*: **segment limit set too high**

Explanation: The specified limit on the /SEGMENTS option is greater than 3072.

Action: Retry with a limit in the range 1 to 3072.

L1009 *option*: **CPARMAXALLOC : illegal value**

Explanation: The number you specified in the /CPARMAXALLOC option is not in the range 1 to 65535.

Action: Retry with a number in the specified range.

L1020 **no object modules specified**

Explanation: You did not specify any object file names to the linker.

Action: Restart LINK, including an object file name.

L1021 **cannot nest response files**

Explanation: A response file has been named within another response file. You have used *@filename* within the response file. The @ symbol is reserved by LINK to signify a response file name.

Action: Edit the response file.

L1022 **response line too long**

Explanation: A line in an automatic response file is longer than 127 characters.

Action: Edit the line.

L1023 **terminated by user**

Explanation: You pressed Ctrl + C or Ctrl + Break.

Action: Your action has terminated LINK. Retry if necessary.

L1024 **nested right parentheses**

Explanation: You typed the contents of an overlay incorrectly on the command line.

Action: Restart LINK, ensuring correct command line syntax.

L1025 **nested left parentheses**

Explanation: You typed the contents of an overlay incorrectly on the command line.

Action: Restart LINK, ensuring correct command line syntax.

L1026 **unmatched right parenthesis**

Explanation: A right parenthesis is missing from the contents specification of an overlay on the command line.

Action: Restart LINK, ensuring correct command line syntax.

L1027 **unmatched left parenthesis**

Explanation: A left parenthesis is missing from the contents specification of an overlay on the command line.

Action: Restart LINK, ensuring correct command line syntax.

L1030 missing internal name

Explanation: You have not specified an internal name for an import in the module definition file.

Action: Give an internal name, so that LINK can identify references to the import.

L1031 module description redefined

Explanation: You have used the DESCRIPTION keyword for a module in the module definition file more than once.

Action: Edit the module definition file, deleting the extra descriptions.

L1032 module name redefined

Explanation: You have defined a module name more than once with the NAME or LIBRARY keyword in the module definition file.

Action: Edit the module definition file, checking the module name definitions.

L1040 too many exported entries

Explanation: You have tried to export more than 3072 names.

Action: Retry with fewer names.

L1041 resident-name table overflow

Explanation: The total length of all your resident-names, together with an overhead of three bytes for each name, is greater than 65534.

Action: Reduce the number or the length of your resident names.

L1042 nonresident-name table overflow

Explanation: The total length of all your nonresident-names, together with an overhead of three bytes for each name, is greater than 65534.

Action: Reduce the number or the length of your nonresident-names.

L1043 relocation table overflow

Explanation: There are more than 65536 load-time relocations for a single segment.

Action: Reduce the number of relocations in the source files and recompile them.

L1044 imported-name table overflow

Explanation: The total length of all your imported-names, together with an overhead of one byte for each name, is greater than 65534 bytes.

Action: Reduce the number or the length of your imported-names.

L1045 too many TYPDEF records

Explanation: An object module contains more than 255 TYPDEF records. These records describe communal variables. This error can only appear with programs produced by compilers (such as the IBM C compiler) that support communal variables.

Action: Reduce the number of TYPDEF records.

L1046 **too many external symbols in one module**

Explanation: An object module specifies more than the limit of 1023 external symbols.

Action: Break the module into smaller parts.

L1047 **too many group, segment, and class names in one module**

Explanation: The program module contains too many group, segment, and class names.

Action: Reduce the number of groups, segments, or classes, and re-create the object files.

L1048 **too many segments in one module**

Explanation: An object module has more than 255 segments.

Action: Split the module or combine some segments.

L1049 **too many segments**

Explanation: The program has more than the maximum number of segments. The /SEGMENTS option specifies the maximum allowed number; the default is 128.

Action: Restart LINK using the /SEGMENTS option with an appropriate number of segments.

L1050 **too many groups in one module**

Explanation: LINK found more than 21 group definitions (GRPDEF) in a single module.

Action: Reduce the number of group definitions or split the module.

L1051 **too many groups**

Explanation: The program defines more than 20 groups in addition to DGROUP.

Action: Reduce the number of groups.

L1052 **too many libraries**

Explanation: You tried to link with more than 32 libraries.

Action: Combine libraries, or use modules that require fewer libraries.

L1053 **symbol table overflow**

Explanation: The program has more symbolic information, such as public, external, segment, group, class, and file names, than the amount that could fit in available real memory.

Action: Combine modules or segments and recreate the object files. Eliminate as many public symbols as possible.

L1054 **out of memory: reduce # in /SEGMENTS:# or /MAP:#**

Explanation: LINK does not have enough memory to allocate tables describing the number of segments requested. The requested limit is either the default of 128 or the value specified with the /SEGMENTS option.

Action: Try linking again using the /SEGMENTS option to select a smaller number of segments (for example, use 64 if the default was used previously), or free some memory by eliminating resident programs or shells.

L1056 too many overlays

Explanation: The program defines more than 63 overlays.

Action: Reduce the number of overlays.

L1057 data record too large

Explanation: A LEDATA record (in an object module) contained more than 1024 bytes of data. This is a translator (compiler or assembler) error.

Action: Note which translator (compiler or assembler) produced the incorrect object module and the circumstances, and contact your IBM Dealer.

L1070 segment size exceeds 64K

Explanation: A single segment contains more than 64KB of code or data. This could be because you attempted to combine identically named segments.

Action: Try compiling (or assembling) and linking using a larger memory model.

L1071 segment _TEXT larger than 65520 bytes

Explanation: This error is likely to occur only in small-model C programs, but it can occur when any program with a segment named _TEXT is linked using the /DOSSEG option of the LINK command. Small-model C programs must reserve code addresses 0 and 1; the reserve is increased to 16 for alignment purposes.

Action: Make the program source code smaller, or change to a larger memory model.

L1072 common area longer than 65536 bytes

Explanation: The program has more than 64KB of communal variables. This error cannot appear with object files produced by the IBM Macro Assembler. It can occur only with programs produced by compilers (such as the IBM C compiler) that support communal variables.

Action: Rewrite your program using fewer or smaller communal variables.

L1073 file-segment limit exceeded

Explanation: There are more than 255 physical or file segments.

Action: Reduce the number of physical or file segments. You could use the /PACKDATA option described in Chapter 7, "LINK Options."

L1074 *name*: **group larger than 64K bytes**

Explanation: A group contains segments that total more than 65536 bytes.

Action: Reduce the number of segments.

L1075 entry table larger than 65535 bytes

Explanation: You have exceeded a linker table size limit because of an excessive number of entry names.

Action: Reduce the number of names in the modules that you are linking.

L1080 **cannot open list file**

Explanation: The disk or the root directory is full, or an invalid file name was specified.

Action: Check that the file name specified is correct. Delete or move files to make space and restart LINK.

L1081 **out of space for run file**

Explanation: The disk on which the .EXE file is being written is full.

Action: Delete or move files to make space and restart LINK.

L1082 **stub .EXE file not found**

Explanation: The stub file specified in the module definition file could not be found.

Action: Check that the correct path to the stub file has been specified.

L1083 **cannot open run file**

Explanation: The disk or the root directory is full.

Action: Delete or move files to make space and restart LINK.

L1084 **cannot create temporary file**

Explanation: The disk or root directory is full.

Action: Delete or move files to make space and restart LINK.

L1085 **cannot open temporary file**

Explanation: The disk or the root directory is full.

Action: Delete or move files to make space and restart LINK.

L1086 **scratch file missing**

Explanation: This is an internal error.

Action: Note the conditions when the error occurs and contact your IBM Dealer.

L1087 **unexpected end-of-file on scratch file**

Explanation: The disk with the temporary linker output file has been removed.

Action: Restart LINK. The temporary disk file is described in Chapter 6, "The LINK Utility."

L1088 **out of space for list file**

Explanation: The disk on which the listing file is being written is full.

Action: Delete or move files to make space and restart LINK.

L1089 *filename*: **cannot open response file**

Explanation: LINK cannot find the specified response file. This usually indicates a typing error.

Action: Include the drive specifier or path, or both, for the response file.

L1090 **cannot reopen list file**

Explanation: You did not replace the original disk when prompted.

Action: Restart LINK.

L1091 **unexpected end-of-file on library**

Explanation: The disk containing the library has probably been removed.

Action: Replace the disk containing the library and restart LINK.

L1092 **cannot open module definitions file**

Explanation: The specified module definition file cannot be opened, or an invalid file name was specified.

Action: Check that the specified file name is correct. Include the drive specifier or path, or both, for the module definition file.

L1093 **object not found**

Explanation: LINK could not open the object module you specified.

Action: Specify full path name or directory in which object module resides.

L1100 **stub .EXE file invalid**

Explanation: The stub file specified in the module definition file is not a valid .EXE file.

Action: Ensure that the stub file is an executable file.

L1101 **invalid object module**

Explanation: One of the object modules was incorrectly formed during compilation.

Action: Recompile your source code. If the error persists, contact your IBM Dealer.

L1102 **unexpected end-of-file**

Explanation: A nonvalid format for a library was found.

Action: Restore the library file from your backup disk and restart LINK.

L1103 **attempt to access data outside segment bounds**

Explanation: A data record in an object module specified data extending beyond the end of a segment. This is a translator error.

Action: Note which translator (compiler or assembler) produced the incorrect object module and the circumstances, and contact your IBM Dealer.

L1104 *filename*: **not valid library**

Explanation: The specified file is not a valid library file. This error causes LINK to stop running.

Action: Ensure that the named file is a valid library file and restart LINK.

L1113 **unresolved COMDEF; internal error**

Explanation: This is an internal error.

Action: Note the conditions when the error occurs and contact your IBM Dealer.

L1114 **file not suitable for /EXEPACK; relink without**

Explanation: For the linked program, the size of the packed load image plus the packing overhead is larger than that of the unpacked load image.

Action: Restart LINK without the /EXEPACK option.

L1115 conflicting iopl-parameters-words value

Explanation: The number of parameter words from a function declared with the _export attribute does not match the number declared in the .DEF file for that function.

Action: Compare the module definition file's _export attribute's numeric value to the number of parameters associated with the specified function.

L2000 imported entry point

Explanation: A MODEND, or starting address record, referred to an imported name. Imported program-starting addresses are not supported.

Action: The starting address record must refer to a nonimported name. Edit the source file.

L2001 fixup(s) without data

Explanation: A FIXUP record occurred without a data record immediately preceding it. This is probably a compiler error.

Action: Note the conditions when the error occurs and contact your IBM Dealer.

L2002 fixup overflow near *number* **in frame seg** *segname* **target seg** *segname* **target offset** *number*

Explanation: The following conditions can cause this error:

- A group is larger than 64KB.
- The program contains an intersegment short jump or intersegment short call.
- The name of a data item in the program conflicts with that of a subroutine in a library included in the link.
- An EXTRN declaration in an assembler language source file appeared inside the body of a segment, as in the following example:

```
code    SEGMENT public  'CODE'
        EXTRN   main:far
start   PROC    far
        call    main
        ret
start   ENDP
code    ENDS
```

The following construction is preferred:

```
        EXTRN   main:far
code    SEGMENT public  'CODE'
start   PROC    far
        call    main
        ret
start   ENDP
code    ENDS
```

Action: Revise the source file and re-create the object file.

L2003 intersegment self-relative fix-up

Explanation: An intersegment self-relative fix-up is not allowed.

Action: Edit the source file.

L2004 **LOBYTE-type fixup overflow**

Explanation: A LOBYTE fixup produced an address overflow.

Action: No action.

L2005 **fixup type unsupported**

Explanation: A fixup type occurred that is not supported by LINK. This is probably a compiler error.

Action: Note the conditions when the error occurs and contact your IBM Dealer.

L2010 **too many fixups in LIDATA record**

Explanation: There are more fixups applying to a LIDATA record than will fit into the LINK's 1024 byte buffer. The buffer is divided between the data in the LIDATA record itself and the run-time relocation items. These are 8 bytes each, so the maximum varies from 0 to 128. This is probably a compiler error.

Action: No action.

L2011 *name* **: NEAR/HUGE conflict**

Explanation: There are conflicting NEAR and HUGE attributes for a communal variable. This error can occur only with programs produced by compilers (such as the IBM C compiler) that support communal variables.

Action: Specify only one of these attributes.

L2012 *name* **: array-element size mismatch**

Explanation: A far communal array has been declared with two or more different array-element sizes (for example, an array declared once as an array of characters and once as an array of real numbers). This error cannot occur with object files produced by the IBM Macro Assembler. It occurs only with IBM C and other compilers that support far communal arrays.

Action: Match the definitions and re-create the object module.

L2013 **LIDATA record too large**

Explanation: A LIDATA record in an object module contains more than 512 bytes of data. It is likely that one of your assembler modules contains a complex structure definition or a series of deeply-nested DUP operators. (LIDATA is a DOS term.)

Example: The following structure definition causes this error:

```
alpha   DB      10DUP(11 DUP(12 DUP(13 DUP(...))))
```

Action: Simplify the structure definition and reassemble the module.

L2022 *name* **alias** *internalname*: **export undefined**

Explanation: A name has been directed to be exported but is not defined anywhere.

Action: Edit the source file and define the export.

L2023 *name* **alias** *internalname*: **export imported**

Explanation: An imported name has been directed to be exported. Items that are not in the source file itself cannot be exported.

Action: Edit the source file.

L2024 *name* **: symbol already defined**

Explanation: Your program defined a symbol name that LINK already used for one of its low-level symbols. For example, the linker generates special names for overlay support.

Action: Edit the source file, and choose another name for the symbol.

L2025 *name* **: symbol defined more than once**

Explanation: A symbol has been defined more than once in the object file.

Action: Edit the source file, removing the extra symbol definition.

L2026 **multiple definitions for entry ordinal** *number*

Explanation: More than one entry point name has been assigned to the same ordinal in the module definition file.

Action: Edit the module definition file.

L2027 *name* **: ordinal too large for export**

Explanation: You tried to export more than 3072 names.

Action: Edit the source file.

L2028 **automatic data segment plus heap exceed 64K**

Explanation: The size of DGROUP near data plus the requested heap size is greater than 64KB.

Action: Delete items from DGROUP or change to a larger memory model.

L2029 **unresolved externals**

Explanation: One or more symbols are declared to be external in one or more modules, but they are not publicly defined in any of the modules or libraries. A list of the unresolved external references appears after the message.

Example:

```
_exit in file(s)
 main.obj (main.c)
_fopen in files(s)
 fileio.obj(fileio.c) main.obj(main.c)
```

The name given before "in file(s)" is the unresolved external symbol. On the next line is a list of object modules that have made references to this symbol. This message and the list are also written to the map file, if one exists.

Action: Supply files that will resolve these external calls.

L2030 **starting address not code (use class 'CODE')**

Explanation: You specified a starting address to LINK that is not a CODE segment.

Action: Reclassify the segment to CODE, or correct the starting point.

L4000 **seg disp included**

Explanation: The error is a result of using the LINK /WARNFIXUP option, described in Chapter 7, "LINK Options."

Action: The segment value is seg and the location offset is disp.

L4001 **frame-relative fixup, frame ignored**

Explanation: A fixup occurred with a frame segment different from the target segment where either the frame or the target segment is not absolute. Such a fix-up is meaningless in the OS/2 environment, so the target segment is assumed for the frame segment.

Action: Check that this is acceptable.

L4002 **frame-relative absolute fixup**

Explanation: A fixup occurred with a frame segment different from the target segment where both frame and target segments are absolute. This fixup is processed using base-offset arithmetic, but the warning is issued because the fixup may not be valid in the OS/2 environment.

Action: Check that this is acceptable.

L4010 **invalid alignment specification**

Explanation: The number following the /ALIGNMENT option is not a power of 2, or is not in numerical form.

Action: Restart LINK using a valid number.

L4011 **PACKCODE value exceeding 65500 unreliable**

Explanation: Code segments of length 65501 – 65536 may be unreliable on the 80286 microprocessor.

Action: Do not use these code segments.

L4012 **load-high disables EXEPACK**

Explanation: The options /HIGH and /EXEPACK are mutually exclusive.

Action: Restart LINK using the correct options.

L4013 **invalid option for new-format executable file ignored**

Explanation: If an OS/2 "protect mode" program is being produced, the options /CPARMAXALLOC, /DSALLOCATE, /EXEPACK, /NOGROUPASSOCIATION, and /OVERLAYINTERRUPT are meaningless, and LINK ignores them.

Action: No action is needed.

L4014 **invalid option for old-format executable file ignored**

Explanation: If a DOS executable program is being produced, the options /ALIGNMENT, /NOFARCALLTRANSLATION, /PACKCODE and /PACKDATA are meaningless, and LINK ignores them.

Action: No action is needed.

L4015 **/CODEVIEW disables /EXEPACK**

Explanation: The options /CODEVIEW and /EXEPACK are mutually exclusive.

Action: Restart LINK with only one of these options.

L4020 *name* **: code-segment size exceeds 65500**

Explanation: Code segments of length 65501 – 65536 may be unreliable on the 80286 microprocessor.

Action: Do not use these code segments.

L4021 **no stack segment**

Explanation: The program does not contain a stack segment defined with the STACK combine type. This message should not appear for modules compiled with the IBM C Compiler but it could appear for an assembler-language module. Normally, every program should have a stack segment with the combine type specified as STACK.

Action: You can ignore this message if you have a specific reason for not defining a stack or for defining one without the STACK combine type.

L4022 *name1, name2* **: groups overlap**

Explanation: Two groups are defined such that one starts in the middle of another. This can occur if you defined segments in a module definition file or assembly file and did not correctly order the segments by class.

Action: Edit the source file and reorder the segments in the group.

L4023 *exportname* **: export internal name conflict**

Explanation: An exported name, or its associated internal name, conflict with an already defined public symbol.

Action: Edit the source file using new names.

L4024 *name* **: multiple definitions for export name**

Explanation: The name *name* has been exported more than once with different internal names. All internal names except the first are ignored.

Action: Edit the source file using new names.

L4025 *name* **: import internal name conflict**

Explanation: An imported name, or its associated internal name, is also defined as an exported name. The import name is ignored. The conflict may come from a definition in either the module definition file or an object file.

Action: Edit the source file or module definition file using new names.

L4026 *modulename* **: self-imported**

Explanation: The module definition file directed that a name be imported from the module being produced.

Action: Edit the module definition file.

L4027 *name* **: multiple definitions for import internal-name**

Explanation: An imported name, or its associated internal name, is imported more than once. The imported name is ignored after the first mention.

Action: Check that the name has been defined correctly.

L4028 *name* **: segment already defined**

Explanation: A segment is defined more than once with the same name in the module definition file. Segments must have unique names for LINK. All definitions with the same name after the first are ignored.

Action: Check that the segment has been defined correctly.

L4029 *name* **: DGROUP segment converted to type data**

Explanation: A segment that is a member of DGROUP has been defined as type CODE in a module definition file or object file. This probably happened because a CLASS keyword in a SEGMENTS statement was not given.

Action: Check the module definition file syntax.

L4030 *name* **: segment attributes changed to conform with automatic data segment**

Explanation: The segment named *name* is defined in DGROUP, but the *shared* attribute is in conflict with the *instance* attribute.

Example: The *shared* attribute is NONSHARED and the *instance* attribute is SINGLE, or the *shared* attribute is SHARED and the *instance* attribute is MULTIPLE. The bad segment is forced to have the right *shared* attribute and the link continues. The image is not marked as having errors.

Action: Check that the linker action is acceptable.

L4031 *name* **: segment declared in more than one group**

Explanation: A segment is declared to be a member of two different groups.

Action: Correct the source file and re-create the object files.

L4032 *name* **: code-group size exceeds 65500 bytes**

Explanation: Code segments of length 65501 – 65536 may be unreliable on the 80286 microprocessor.

Action: Do not use these code segments.

L4034 **more than 239 overlay segments; extra put in root**

Explanation: You specified an overlay structure containing more than 239 segments. The extra segments have been assigned to the root overlay, starting with the 234th segment.

Action: Check that this is acceptable.

L4036 **no automatic data segment**

Explanation: The program or dynamic link library did not define a group named DGROUP. This is recognized by LINK as the automatic data segment.

Action: Edit the source file.

L4040 **NON-CONFORMING : obsolete**

Explanation: In the module definition file, NON-CONFORMING is a valid keyword for earlier versions of LINK and is now obsolete.

Action: Check module definition file syntax.

L4041 HUGE segments not supported

Explanation: This feature has not been implemented by LINK.

Action: Edit the source file.

L4042 cannot open old version

Explanation: An old version of the .EXE file, specified with the OLD keyword in the module definition file, could not be opened.

Action: Check the path specification for the executable file.

L4043 old version not segmented-executable format

Explanation: The old version of the .DLL file, specified with the OLD keyword in the module definition file, does not conform to segmented-executable format.

Action: No action.

L4044 minalloc feature is obsolete; ignored

Explanation: A line in the SEGMENTS section of the module definition (.DEF) file contained out-of-date syntax.

Action: Refer to Chapter 8, "Module Definition File Statements."

L4045 <name>:is name of output file

Explanation: A dynamic link library file was created without specifying an extension. In such cases, the linker supplies an extension of .DLL. This is to warn you in case you expected an .EXE file to be generated

Action: No action.

L4046 module name different from output filename

Explanation: You specified a module name by way of the NAME or LIBRARY statement in the definitions file that is different from the output file (.EXE or .DLL) name. This will likely cause problems in BINDING the file or in using it in OS/2 mode.

Action: Rename the file to match the module name before it is executed.

L4050 too many public symbols

Explanation: The /MAP option has been used to request a sorted listing of public symbols in the map file, but there were too many symbols to sort (the default is 2048 symbols). LINK will produce an unsorted listing of the public symbols.

Action: Restart LINK using /MAP:*number*.

L4051 *filename* **: cannot find library**

Explanation: LINK could not find the specified library file.

Action: Enter a new file name, a new path specification, or both.

L4053 VM.TMP : illegal file name; ignored

Explanation: VM.TMP cannot be used for an object file name.

Action: Rename the file and restart LINK.

L4054 *filename* **: cannot find file**

Explanation: LINK could not find the specified file.

Action: Enter a new file name, a new path specification, or both.

Resource Compiler Error Messages

The error messages produced by the resource compiler utility (RC) are listed below. In addition to these, you may encounter messages produced by the IBM C/2 language compiler during the resource compilation process. These are listed in *IBM C/2 Compile, Link, and Run.*

Error Message Descriptions

Accelerator type required (CHAR, SCANCODE, or VIRTUALKEY)

Explanation: An *acceloption* has not been specified in the accelerator table to define the type of accelerator. If the accelerator character code is something other than a single character or a character preceded by a caret (^), an *acceloption* is required.

Action: Check accelerator table syntax.

BEGIN expected in accelerator table

Explanation: BEGIN keyword missing from accelerator table.

Action: Check syntax.

BEGIN expected in dialog or window template

Explanation: BEGIN keyword missing from dialog or window template.

Action: Check syntax.

BEGIN expected in menu

Explanation: BEGIN keyword missing from menu.

Action: Check syntax.

BEGIN expected in message table

Explanation: BEGIN keyword missing from message table.

Action: Check syntax.

BEGIN expected in RCData

Explanation: BEGIN keyword missing from RCData table.

Action: Check syntax.

BEGIN expected in String Table

Explanation: BEGIN keyword missing from string table.

Action: Check syntax.

Cannot re-use message constants

Explanation: Message identifier has been used more than once in message table.

Action: Check message table syntax.

Cannot re-use string constants

Explanation: Message identifier has been used more than once in string table.

Action: Check string table syntax.

Comma expected after item string

Explanation: A comma must be used to separate the menu item identifier and the menu item string.

Action: Check menu syntax.

Control character out of range (^A - ^Z)

Explanation: Accelerator character codes that use the Ctrl key, and are therefore preceded by a caret (^), must use alphabetic keys.

Action: Check accelerator table syntax.

END expected in dialog

Explanation: END keyword missing from dialog template.

Action: Check syntax.

END expected in menu

Explanation: END keyword missing from menu.

Action: Check syntax.

Error creating temp file

Explanation: Temporary files are created by the resource compiler during the compilation process.

Action: Check that there is sufficient disk space to run the resource compiler, and restart the resource compiler.

Expected comma in accelerator table

Explanation: Commas are used in the accelerator table to separate the accelerator key, the accelerator command, and the accelerator options.

Action: Check accelerator table syntax.

Expected ID value for menuitem

Explanation: A selection identifier is needed for each item within a menu.

Action: Check menu syntax.

Expected menu string

Explanation: A character string should be specified in the menu definition to describe the menu selection.

Action: Check menu syntax. The string should be enclosed in double quotation marks.

Expected numeric command value

Explanation: A number should be used in the accelerator table to identify the message that is generated by an accelerator key.

Action: Check accelerator table syntax.

Expected numeric constant in message table

Explanation: The identifier that precedes a message definition must be an integer.

Action: Check message definition syntax.

Expected numeric constant in string table

Explanation: The identifier that precedes a string definition must be an integer.

Action: Check string definition syntax.

Expected numerical dialog constant

Explanation: Integers are required in dialog and window templates to specify the coordinates and dimensions of the dialog box.

Action: Check syntax of dialog box definition.

Expected string in message table

Explanation: A character string was not found in the message table.

Action: Check syntax. The string should be enclosed in double quotation marks.

Expected string in string table

Explanation: A character string was not found in the string table.

Action: Check string table syntax. The string should be enclosed in double quotation marks.

Expected string or constant accelerator command

Explanation: The accelerator character code is missing.

Action: Check accelerator table syntax.

File not found

Explanation: The resource compiler could not find the .RC or .RES file that you requested.

Action: Check that the file is in the current directory and check the path to the directory.

Illegal empty BEGIN/END block found, resource not written

Explanation: A BEGIN/END block with no DIALOG, CONTROL, or WINDOW statements in it was found in the dialog template.

Action: Delete unwanted BEGIN/END blocks.

Invalid accelerator

Explanation: The character code specified as an accelerator key must be a valid keyboard operation.

Action: Check accelerator key definition syntax.

Invalid accelerator option

Explanation: The accelerator option must be a valid keyword.

Action: Check syntax.

Invalid control character

Explanation: The accelerator key definition can include an caret (^) to specify that the key should be used with the Ctrl key.

Action: Check accelerator key definition syntax.

Invalid Type

Explanation: The resource type must be a valid keyword.

Action: Check resource definition syntax.

Non-numeric template ID in dialog or window template

Explanation: The resource identifier must be an integer.

Action: Check dialog or window template syntax.

Only one top level window allowed

Explanation: Only one DIALOG, CONTROL, or WINDOW statement is allowed within the dialog or window template.

Action: Check dialog or window template syntax.

Resource Type keyword expected

Explanation: The resource type must be specified in the resource script file.

Action: Check resource definition syntax.

String literal too long

Explanation: Strings cannot be longer than 255 characters.

Action: Edit the string.

Text string or ordinal expected in control

Explanation: A text string can be specified in the DIALOG statement of a dialog template to give it a title. If a title is not required, double quotation marks should be used with no characters between them ("").

Action: Edit DIALOG statement.

Unbalanced parentheses

Explanation: The left and right parentheses have not been matched.

Action: Edit the parentheses.

Undefined keyword or key name

Explanation: An invalid keyword or key name has been used.

Action: Check syntax.

Unexpected end of file in string literal

Explanation: The double quotation marks have not been closed at the end of a character string.

Action: Edit the string.

Unexpected value in RCData

Explanation: The variable defined in RCData must be a string or a number.

Action: Check the RCData syntax.

Unknown dialog or window token

Explanation: The dialog and window templates must use only the DIALOG, WINDOW, or CONTROL keywords.

Action: Check the dialog or window template syntax.

Unknown menu subtype

Explanation: Items within a menu can be specified only with the MENUITEM and SUBMENU keywords.

Action: Check menu definition syntax.

BIND Error Messages

BIND terminated by user

Explanation: Ctrl + C or Ctrl + Break pressed during execution.

Action: Restart BIND.

Cannot create file: *xxxxxxxx*

Explanation: Cannot create either the temporary file or the .BM file.

Action: Delete or move files to make space and restart BIND.

Could not find link in your path

Explanation: Unable to find LINK.EXE for execution of intermediary link.

Action: Change the path statement, or copy the linker into the existing path. Restart BIND.

Duplicate global name: *xxxxxxxx*

Explanation: Multiple definitions of a global name.

Action: Check global name definitions.

Duplicate in file name given: *xxxxxxxx*

Explanation: File to be run through BIND named in multiple locations on the command line.

Action: Check command line.

Error during link of file, link error status: *xxxxxxxx*

Explanation: Errors were encountered while resolving dynamic link references with API.LIB and any user-supplied API support libraries. This error includes unresolved references, lack of memory, and disk I/O errors.

Action: See Chapter 6, "The LINK Utility" for more information on linking.

Internal error – Lname not found: *xxxxxxxx*

Explanation: An internal error occurred while running BIND.

Action: Try your backup copy of BIND.

Internal error – System call (*xxxx*) **return error**

Explanation: An internal error occurred while running BIND.

Action: Try your backup copy of BIND.

It is unsafe to bind an executable linked with PROTMODE.

Explanation: If there is floating point code in the executable, it will not run reliably in real mode. However, if the executable contains no floating point code, it will run reliably in both real mode and protect mode.

Action: Only run executables that have floating point code and have been linked with PROTMODE in protect mode.

No infile>.EXE<, execute with no parameters for usage

Explanation: You did not provide the name of the executable file to be used by BIND.

Action: Restart BIND.

No outfile, execute with no parameters for usage

Explanation: You set the flag (/o) for an alternative DOS executable file, but did not provide a name for it.

Action: Restart BIND.

Out of memory

Explanation: There was not enough memory to run BIND.

Action: Reduce the number of programs running in your system and restart BIND.

Resource tables not supported

Explanation: The OS/2 .EXE header refers to names in the resource table.

Action: Delete these references from the .DEF file and rebuild the .EXE file.

Structure error in .EXE file

Explanation: The .EXE file was in error.

Action: Rebuild the .EXE file and restart BIND.

Structure error in .LIB file: *xxxxxxxx*

Explanation: The .LIB file used was not of an expected format.

Action: Rebuild the .LIB file using IMPLIB and restart BIND.

This file is not a well-formed .EXE

Explanation: The file to be bound was not a correct OS/2 .EXE file.

Action: Rebuild the .EXE file, and restart BIND.

Too many libraries specified, *xx* allowed

Explanation: Too many .LIB files were specified.

Action: Combine the .LIB files and restart BIND.

Unknown flag: *xxxx*

Explanation: You passed an unknown command line flag (option).

Action: The only options that are allowed are /m, /n, and /o. Restart BIND.

Unrecoverable I/O error

Explanation: A disk error occurred while running BIND.

Action: Run CHKDSK on the disk and restart BIND.

IMPLIB Error Messages

usage: Implib *xxxxxxxx xxxxxxxx*

Explanation: The command was entered incorrectly.

Action: Restart IMPLIB. Refer to Chapter 5, "Building a Dynamic Link Library" for the correct syntax.

xxxx **near line** *xxxx* **in definitions file**

Explanation: There was an error in the module definition (.DEF) file.

Action: Correct the symbol shown in the message, at the line number given. Restart IMPLIB.

Implib: *xxxxxxxx* **multiply defined**

Explanation: An export name was repeated within or across the module definition (.DEF) files.

Action: Eliminate duplicate definitions of the export name.

Implib: Cannot create *xxxxxxxx*

Explanation: IMPLIB was unable to open or create the target library specified.

Action: Check the file name and available space. Restart IMPLIB.

Implib: Cannot open *xxxxxxxx*

Explanation: IMPLIB was unable to open one of the specified input module definition (.DEF) files.

Action: Check the file name. Restart IMPLIB.

implib: Out of heap space

Explanation: There was not enough memory available to run IMPLIB.

Action: Reduce the number of programs presently running in your system and restart IMPLIB.

implib: Out of space on input file

Explanation: There was not enough disk space available to create the target library.

Action: Delete or move files to make space on the disk and restart IMPLIB.

MKMSGF Error Messages

MKMSGF: Error writing output file

Explanation: Error during output to target file.

Action: Make sure there is sufficient disk space or that the drive is ready. Retry the command.

MKMSGF: Error reading input file

Explanation: Error during input from source file.

Action: Make sure the source message file exists and that the drive is ready. Retry the command.

MKMSGF: File not found

Explanation: Input file could not be found

Action: Retry the command, using the correct source message file name.

MKMSGF: Insufficient storage

Explanation: Not enough storage to execute program or too many messages in the file. Message limit is about 6000.

Action: Reduce the number of programs running in your system. Or reduce the size of the message file by either deleting messages or by having shorter messages. Retry the command.

MKMSGF: Invalid message file format

Explanation: Input file is not a recognizable message text file.

Action: If an incorrect file name was entered, retry the command with the correct source message file name. Otherwise, see the *Programming Guide*.

MKMSGF: Message ID out of sequence

Explanation: A message was detected that was out of the required sequential order.

Action: Correct the error by editing your source message file and renumbering the messages. You may also want to delete or insert the appropriate message numbers to achieve the required sequential order.

MKMSGF: Message XXXX too long

Explanation: The message was too long to be processed (limit is approximately 2K characters).

Action: Correct the error by editing your source message file and making the message shorter. Then, retry the command.

MKMSGF: Syntax error
MKMSGF infile[.ext] outfile [.ext]

Explanation: User entered the command incorrectly. Proper syntax is displayed to user.

Action: Retry the command using the proper syntax.

MSGBIND Error Messages

usage: MSGBIND scriptfile

Explanation: The command was entered incorrectly.

Action: Restart MSGBIND. Refer to Chapter 4, "Building Text Window and Full-Screen Applications" for the correct syntax.

Reading messages from *xxxxxxxx*

Explanation: Messages from the displayed message file are being read into memory.

Action: None. This message is for information only.

Reading messages from *xxxxxxxx* **—file not found**

Explanation: The message file was not found in the path specified in the input file.

Action: Edit the input file, correcting the path or file name, or both. Restart MSGBIND.

Reading messages from *xxxxxxxx* **—not created with MKMSGF utility**

Explanation: The message file displayed was not created using the MKMSGF utility.

Action: Convert the source message file to a formatted message file using the MKMSGF utility, and restart MSGBIND. See Chapter 4, "Building Text Window and Full-Screen Applications" for information on MKMSGF utility.

Reading messages from *xxxxxxxx* **—not enough memory**

Explanation: There was not enough memory available to store the messages.

Action: Reduce the number of programs running in your system, and restart MSGBIND.

Updating *xxxxxxxx*

Explanation: The .EXE file name displayed is being updated with the requested messages.

Action: None. This message is for information only.

Updating *xxxxxxxx* **—file not found**

Explanation: The .EXE file name was not found in the path specified in the input file.

Action: Edit the input file, correcting the path and r file name, or both. Restart MSGBIND.

Updating *xxxxxxxx* – not linked with MSGSEG.OBJ

Explanation: The .EXE file name displayed made no DosGetMessage function calls, or the .EXE file was not linked with DOSCALLS.LIB. Messages can only be bound to applications that make message retriever function calls, such as DosGetMessage.

Action: If the .EXE file does make message retriever function calls, relink the application using DOSCALLS.LIB, and restart MSGBIND.

Writing messages

Explanation: The .EXE file is being updated with the messages requested.

Action: None. This message is for information only.

MSGBIND: I/O error seeking infile

Explanation: A disk error occurred while seeking either the message file or the .EXE file.

Action: Run CHKDSK on the drive containing the file and restart MSGBIND.

MSGBIND: I/O error writing file

Explanation: A disk error occurred while writing messages to the .EXE file.

Action: Run CHKDSK on the drive containing the .EXE file, and restart MSGBIND.

MSGBIND: Must specify .EXE file before message file

Explanation: The input file was in error.

Action: Refer to Chapter 4, "Building Text Window and Full-Screen Applications" for the correct input file format.

MSGBIND: Must specify message file before message number

Explanation: The input file was in error.

Action: Refer to Chapter 4, "Building Text Window and Full-Screen Applications" for the correct input file format.

MSGBIND: Out of memory, needed *xxxx* bytes

Explanation: There was not enough memory available to run MSGBIND.

Action: Reduce the number of programs presently running in your system and restart MSGBIND.

MSGBIND: Premature EOF during copy

Explanation: The .EXE file was not built correctly.

Action: Rebuild the .EXE file and restart MSGBIND.

MSGBIND: Unable to create temp file – MSGBIND.TMP

Explanation: An error occurred while creating the intermediary file MSGBIND.TMP.

Action: Delete or move files to make disk space available. If MSGBIND.TMP is present as a read-only file, it must first be deleted. Restart MSGBIND.

MSGBIND: Unable to open *xxxxxxxx*

Explanation: The input file specified was not found or an error occurred when opening the message file.

Action: Restart MSGBIND using the correct input file name or a backup copy of the message file.

WARNING: Skipping messages for this file

Explanation: The .EXE file was in error, so the messages were not bound to it. Either the .EXE file does not exist, or it has not been linked with DOSCALLS.LIB.

Action: If the .EXE file name is correct, relink the application using DOSCALLS.LIB, and restart MSGBIND.

WARNING: Skipping message numbers for this file

Explanation: The message file was in error, so all messages from this message file were ignored. The message file may not exist, may not be formatted correctly, or there may not be enough memory to store all the messages.

Action: If the message file name is correct and has been correctly formatted, check the memory available for the file. Restart MSGBIND.

WARNING: *xxxx* is an invalid message number

Explanation: The message number specified was not found in the message file.

Action: Edit the input file and correct any message numbers that are in error. Restart MSGBIND.

Index

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Numerics

Special Characters

IBM United Kingdom
International Products Limited
PO Box 41, North Harbour
Portsmouth, PO6 3AU
England

Printed in Denmark
Rønsholt Trykkeri A/S